Títol: Il·luminació global d'entorns urbans
Autor: Óscar Argudo Medrano

Data: 10 de juny de 2011

Director: Carlos Andújar Gran
Co-director: Gustavo Patow
Departament: Llenguatges i Sistemes Informàtics (LSI)

Índex

Índex de figures

Índex de taules

Part I

Introducció, conceptes i treball previ

Capítol 1

Introducció

Al llarg dels últims anys, han aparegut diversos projectes i eines que ofereixen vistes de mapes o de satèl·lit de diverses ciutats del món, com per exemple Google Earth[1], Microsoft Bing Maps[2] o Nokia Ovi Maps[3]. Més recentment, aquestes eines citades han incorporat la visualització interactiva en tres dimensions de moltes de les principals ciutats del món.

Aquesta visualització ha anat evolucionant per oferir cada vegada més detalls: per exemple, a les eines de Google, els edificis que inicialment es mostraven com blocs blancs ara incorporen textures extretes de fotografies a peu de carrer. No obstant, aquesta visualització encara es podria millorar afegint el realisme que aporta la il·luminació del model basada en models globals. Aquest projecte pretén aportar aquest realsme a la visualització de les ciutats.

1.1 Objectius del projecte

L'objectiu principal del projecte és aplicar un algorisme d'il·luminació global a models de ciutats basats en dades reals d'aquestes. Es volen obtenir imatges resultants el més realistes possibles.

A més, ens agradaria que la generació d'aquestes imatges fos suficientment ràpida per a permetre les visualitzacions en temps interactiu (entre 5 i 20 imatges per segon) o fins i tot en temps real (més de 25 imatges per segon).

Per aconseguir els dos objectius anteriors, partirem d'algun algorisme conegut i el modificarem per adaptar-lo a les característiques específiques dels models urbans. Fent aquestes optimitzacions per al tipus concret de model, esperem aconseguir millors resultats que amb l'algorisme original.

1.2 Organització de la memòria

La primera part d'aquesta memòria correspon a la introducció, conceptes i treballs previs. L'objectiu és, d'una banda, apropar al lector a l'àrea de la generació de gràfics realistes i, a la vegada, reflectir la feina realitzada de lectura i investigació prèvia al disseny del nostre algorisme. Per tant, el capítol 2 començarà definint la base física que hi ha darrere dels algorismes descrits al capítol 3. Tot seguit, s'explicarà en detall un d'aquests algorismes al capítol 4, ja que serà el que agafarem com a base inicial. Finalment, al capítol 5 es

[1] http://earth.google.com
[2] http://maps.bing.com
[3] http://maps.ovi.com

veurà quins programes i entorns hi ha actualment que permetin implementar algorismes d'il·luminació global amb resultats interactius.

La segona part de la memòria és el desenvolupament tècnic del projecte, que en aquest cas és en gran mesura la descripció de l'algorisme que s'ha dissenyat i els resultats obtinguts. El capítol 6 fa un breu resum general de l'algorisme, per a poder entendre posteriorment millor el per què d'alguns detalls quan s'expliquin als capítols 7, 8, 9, 10 i 11, que descriuen el preprocés i els diferents passos de l'algorisme. Els resultats s'analitzen al capítol 13. Els aspectes relatius al disseny del programa i a la planificació del projecte es poden trobar als capítols 12 i 14, respectivament.

Finalment, als annexos es troba una breu descripció d'OpenGL (annex A) i de CUDA (annex B), que pot ser d'ajuda als lectors no familiaritzats amb els termes utilitzats. De la mateixa manera, l'annex D recull les definicions més rellevants sobre il·luminació global que s'han fet al llarg de la memòria, per a consultar-les ràpida i còmodament en cas de necessitat. També s'inclou un manual d'usuari del visualitzador a l'annex C.

Capítol 2

Nocions sobre física de la llum

Per a poder entendre el funcionament dels algorismes de generació d'imatges físicament realistes, abans caldrà veure alguns conceptes físics sobre la llum, la seva propagació i la interacció amb els materials. No entrarem gaire en els detalls físics, sinó que en farem una breu descripció com la que podem trobar als llibres de referència sobre gràfics realistes [12, 20, 2, 40].

2.1 Magnituds radiomètriques

La radiometria és la branca de la física encarregada de l'estudi de la mesura de la radiació electromagnètica. El seu camp cobreix tot l'espectre de longituds d'ona, mentre que la fotometria se centra en la part visible de l'espectre i la percepció per l'ull humà.

Segons el model quàntic de la llum, aquesta està formada per fotons, l'energia dels quals depèn de la longitud d'ona λ i ve donada per l'expressió $e_\lambda = \frac{h\,c}{\lambda}$, on $h \approx 6.63 \cdot 10^{-34}\,\mathrm{J\,s}$ és la constant de Planck, i $c = 299\,792\,458\,\mathrm{m/s}$ és la velocitat de la llum al buit. Si volem calcular l'**Energia radiant** (Q) d'una col·lecció de fotons sobre tot l'espectre a partir de la distribució n_λ de fotons segons la longitud d'ona, farem:

$$Q = \int_0^\infty n_\lambda\, e_\lambda\, d\lambda$$

La següent magnitud que podem definir és la **Potència radiant** o **Flux**, que denotarem Φ i es mesura en Watt (W). És la quantitat total d'energia que travessa una superfície per unitat de temps, és a dir, $\Phi = \frac{dQ}{dt}$. És important notar que el flux no ens informa sobre la distribució d'aquesta energia sobre la superfície. Per exemple, podem dir que una llum emet 50 W o que sobre una taula incideixen 20 W.

Del flux en podem derivar dues més. Si volem representar la distribució direccional del flux, farem servir la **Intensitat radiant** (I), que es defineix com el flux per unitat d'angle sòlid ω mesurat en estereoradians:

$$I(\vec{\omega}) = \frac{d\Phi}{d\vec{\omega}}$$

Alternativament, podem estar interessats en la distribució del flux sobre la superfície, sense importar la direcció d'aquest. Per tant, es defineix la densitat en àrea del flux radiant i es mesura en $\mathrm{W/m^2}$. Normalment es diferencia entre la **Irradiància** (E), que és el flux incident per unitat de superfície, i la **Emitància radiant**, que és el flux emès per unitat

de superfície. Aquesta última magnitud també és coneguda com **Radiositat** (B).

$$E(x) = \frac{d\Phi}{dA} \qquad M(x) = B(x) = \frac{d\Phi}{dA}$$

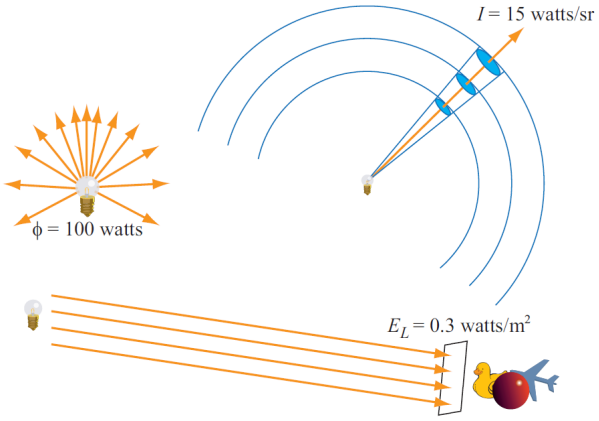

Figura 2.1: Representació del flux emès per la bombeta, la intensitat en una direcció donada i la irradiància arribant a una superfície. *Imatge extreta de [2].*

Finalment, definim la **Radiància** (L) com el flux per unitat d'angle sòlid i de superfície projectada. És a dir, ens informa sobre la distribució sobre la superfície i en funció de la direcció. Es mesura en $\mathrm{W}/(\mathrm{m}^2\, sr)$. L'expressió és:

$$L(x, \vec{\omega}) = \frac{d^2\Phi}{d\vec{\omega}\, dA^{\perp}} = \frac{d^2\Phi}{d\vec{\omega}\, dA\, \cos\theta}$$

Fixem-nos que l'àrea que es fa servir és la projectada sobre el pla perpendicular a la direcció del raig, com mostra la figura 2.2. Intuïtivament, podem veure que com més paral·lels siguin els rajos amb la superfície, el flux es repartirà sobre una àrea major. També, observem que és una magnitud 5-dimensional, ja que tenim tres graus de llibertat per a la posició i dos per a la direcció.

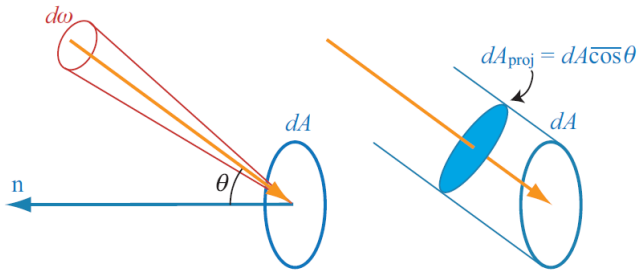

Figura 2.2: Radiància: flux per unitat d'angle sòlid i de superfície projectada. *Imatge extreta de [2].*

La radiància és, probablement, la magnitud radiomètrica més important pels algorismes de síntesi d'imatges realistes, ja que captura l'aparença dels objectes a l'escena. A més a més, compleix dues propietats que la fan ser una candidata perfecta per al càlcul de la il·luminació:

1. És invariant al llarg de camins rectilinis si no hi ha medis participatius. És a dir, la radiància que surt del punt x cap al punt y és la mateixa que arriba al punt y provinent del punt x si assumim que viatja pel buit per a no interferir amb el medi. Matemàticament:

$$L(x \rightarrow y) = L(y \leftarrow x)$$

2. La resposta dels sensors (càmeres o l'ull humà) és proporcional a la radiància incident sobre ells, amb una constant de proporcionalitat que depèn de la geometria del sensor. Juntament amb la propietat anterior, explica per què el color o brillantor que percebem d'un objecte no varia amb la distància.

També és fàcil veure que a partir de la radiància podem trobar la resta de magnituds radiomètriques:

$$\Phi = \int_A \int_\Omega L(x, \vec{\omega}')(\vec{n} \cdot \vec{\omega}') d\vec{\omega}' dA_x = \int_A \int_\Omega L(x \to \Theta) \cos\theta d\omega_\Theta dA_x$$

$$E(x) = \int_\Omega L(x \leftarrow \Theta) \cos\theta d\omega_\Theta$$

$$B(x) = \int_\Omega L(x \to \Theta) \cos\theta d\omega_\Theta$$

Fent servir la notació $L(x \to \Theta)$ per a la radiància sortint de x en direccio Θ i $L(x \leftarrow \Theta)$ per a la radiància entrant a x des de la direcció Θ.

Símbol	Magnitud	Unitats
Q	Energia radiant	J
Φ	Flux o potència radiant	W
I	Intensitat radiant	W / sr
E	Irradiància (incident)	W / m^2
M	Emitància radiant (sortint)	W / m^2
B	Radiositat (sortint)	W / m^2
L	Radiància	W / ($m^2 \cdot$ sr)
L_λ	Radiància espectral	W / ($m^2 \cdot$ sr $\cdot$ nm)

Taula 2.1: Resum de les diferents magnituds radiomètriques, símbols i unitats de mesura.

2.1.1 Relació entre radiometria i fotometria

Com hem comentat, la fotometria té en compte la resposta visual d'un observador estàndard. Es defineix una corba de resposta segons la longitud d'ona al llarg de l'espectre visible i es ponderen les magnituds en funció d'aquesta corba. Així, per exemple, es té en compte que l'ull humà percep més brillant la llum verda que no pas la vermella o blava. Podríem definir l'**energia lluminosa** (Q_v) com:

$$Q_v = \int_{\text{visible}} Q_\lambda V(\lambda) d\lambda$$

on $V(\lambda)$ és la resposta visual a la longitud d'ona λ per a l'observador estàndard. Es poden derivar la resta de magnituds de la mateixa manera que hem fet abans. La taula 2.2 mostra els noms i les unitats per a les diverses magnituds fotomètriques.

L'avantatge de la relació entre radiometria i fotometria és que podem fer els càlculs radiomètrics per a la nostra escena i després, com a post-procés, afegir la resposta visual d'un observador. Aquest procés és conegut amb el nom de *tone mapping*.

Magnitud fotomètrica	Símbol	Unitats	Magnitud radiomètrica
Quantitat de llum / energia lluminosa	Q_v	talbot = lm · s	Energia radiant
Flux lluminós / potència lluminosa	F	lumen (lm)	Flux o potència radiant
Intensitat lluminosa	I_v	candela (cd)	Intensitat radiant
Luminància	E_v	lux (lx)	Irradiància
Emitància lluminosa	M_v	lux (lx)	Emitància radiant / Radiositat
Luminància	L_v	nit = $\mathrm{cd/m^2}$	Radiància

Taula 2.2: Relació entre les magnituds i unitats de la radiometria i la fotometria.

2.2 Interacció de la llum amb les superfícies

La llum emesa per les fonts de llum de la nostra escena es propaga a través d'ella i pot trobar-se amb objectes. Quan es troba amb la superfície d'un objecte, aquesta llum pot ser reflectida, transmesa a través de l'objecte o absorbida per aquest. Per a simplificar, assumirem que aquestes interaccions no afecten a la longitud d'ona de la llum, i per tant estarem deixant a part efectes com la fluorescència o fosforescència.

2.2.1 Funcions BSSRDF i BRDF

Quan un raig de llum arriba a un material, normalment entra lleugerament dins la superfície, es va dispersant per les partícules d'aquesta i acaba sortint per un altre punt. Aquest fenomen, conegut com *subsurface scattering*, és molt evident per materials com la cera, el marbre o la pell, però passa en major o menor grau amb tots els materials. Matemàticament, aquesta interacció és modelada per la funció BSSRDF, per les sigles en anglès de *Bidirectional Scattering Surface Reflectance Distribution Function*. Aquesta funció S ens relaciona un diferencial de radiància reflectida dL_r a x en la direcció Θ amb el diferencial de flux incident $d\Phi_i$ al punt x' des de la direcció Ψ:

$$S(x, \Theta, x', \Psi) = \frac{dL_r(x \to \Theta)}{d\Phi_i(x' \leftarrow \Psi)}$$

La BSSRDF és el model més general del transport de la llum a les superfícies. No obstant, podem veure que és una funció 8-dimensional i és molt costosa d'avaluar. Per això, a no ser que vulguem tractar algun material com els que hem comentat on el fenomen de *subsurface scattering* és molt present, normalment s'assumeix que el punt d'incidència i de reflexió de la llum són el mateix (veure figura 2.3). Aquesta aproximació és coneguda com BRDF, que són les sigles de *Bidirectional Reflectance Distribution Function*. Podem definir una BRDF f_r com la relació entre la radiància reflectida i la irradiància:

$$f_r(x, \Psi \to \Theta) = \frac{dL_r(x \to \Theta)}{dE_i(x \leftarrow \Psi)} = \frac{dL_r(x \to \Theta)}{dL_i(x \leftarrow \Psi)\cos(\vec{n}_x, \Psi)\, d\omega_\Psi}$$

La BRDF redueix el problema a una funció 6-dimensional. Per a materials pels quals no tenim en compte la transmissió de la llum, normalment es defineix a l'hemisferi situat sobre x. No obstant, per a materials transparents el podem definir sobre tota l'esfera de direccions (4π estereoradians). En aquest cas, se l'acostuma a anomenar BSDF, de *Bidirectional Scattering Distribution Function*. Si, de manera anàloga a la BRDF, només la definim sobre l'hemisferi corresponent a la transmissió, s'acostuma a dir BTDF, de *Bidirectional Transmittance Distribution Function*.

Una funció BRDF f_r ha de complir les propietats següents:

1. Rang: la BRDF pot prendre qualsevol valor positiu i dependre de la longitud d'ona.

2. Dimensió: per a un punt de la superfície donat, la BRDF és una funció 4-dimensional, dues dimensions per a l'angle incident i dues per al de sortida. En general, una BRDF és *anisotròpica*, és a dir, que rotant la superfície respecte la seva normal podem obtenir valors diferents de f_r. No obstant, molts materials són *isotròpics* i no depenen d'aquesta orientació de la superfície.

3. Reciprocitat de Helmhotz: el valor d'una BRDF no varia si intercanviem l'angle incident Θ i reflectit Ψ. És a dir, que la BRDF és independent del sentit del flux de la llum: $f_r(x, \Theta \rightarrow \Psi) = f_r(x, \Theta \leftarrow \Psi)$. Degut a això, normalment es fa servir la notació $f_r(x, \Theta \leftrightarrow \Psi)$.

4. Relació entre radiància incident i reflectida: el valor de f_r per a una direcció incident específica no depèn de la possible irradiància a altres angles d'incidència. Per tant, es comporta com una funció lineal respecte totes les possibles direccions incidents. Per conèixer la radiància total reflectida degut a una distribució d'irradiància podem integrar sobre l'hemisferi:

$$\begin{aligned} L_r(x \rightarrow \Theta) &= \int_{\Omega_x} f_r(x, \Psi \rightarrow \Theta)\, dE(x \leftarrow \Psi) \\ &= \int_{\Omega_x} f_r(x, \Psi \rightarrow \Theta)\, L_i(x \leftarrow \Psi)\, \cos(\vec{n}_x, \Psi)\, d\omega_\Psi \end{aligned}$$

5. Conservació de l'energia: la llei de conservació de l'energia implica que el total de flux reflectit en totes les direccions ha de ser igual o menor que el total de flux incident des de totes les direccions. Per a que això sigui cert, es pot veure que s'ha de complir la condició següent:

$$\forall \Psi : \int_{\Omega_x} f_r(x, \Psi \rightarrow \Theta)\, L_i(x \leftarrow \Psi)\, \cos(\vec{n}_x, \Psi)\, d\omega_\Psi \leq 1$$

Aquesta expressió és condició necessària i suficient per a la conservació de l'energia. Per tant, s'acostuma a fer servir per a verificar si un determinat model de BRDF és correcte físicament.

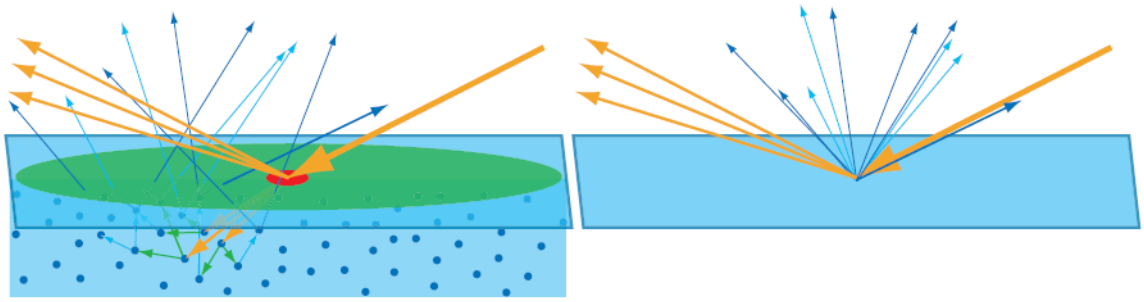

Figura 2.3: A l'esquerra: exemple d'interacció on el punt de sortida de la llum és diferent del d'entrada, es podria modelar amb una BSSRDF. A la dreta, la BRDF resultant si assumim que el punt d'entrada i de sortida és el mateix. *Imatge extreta de [2].*

2.2.2 Superfícies difoses

Alguns materials reflecteixen la llum de manera uniforme sobre tot l'hemisferi de reflexió, és a dir, que la radiància reflectida és independent de la direcció de reflexió. El valor de la BRDF és constant per a tots els valors de Θ. Per tant, per a un observador, la superfície té el mateix aspecte des de tots els punts de vista. L'expressió d'una BRDF difosa ideal és:

$$f_r(x, \Psi \leftrightarrow \Theta) = \frac{\rho_d}{\pi}$$

on ρ_d s'anomena reflectància i representa la fracció de llum incident que és reflectida a la superfície. Per tant, el valor estarà entre 0 i 1.

2.2.3 Superfícies especulars

Les superfícies especulars són aquelles que reflecteixen o refracten la llum en una direcció específica.

Per a les reflexions especulars, la direcció de reflexió es pot calcular fent servir la **llei de la reflexió**, que afirma que la direcció d'incidència i la de reflexió fan el mateix angle amb la normal de la superfície i que tots tres vectors són coplanars. Per tant, podem trobar aquesta direcció $\vec{R}$ a partir de la incident Ψ com:

$$\vec{R} = 2(\vec{n}_x \cdot \Psi)\vec{n}_x - \Psi$$

Una reflexió especular perfecta és aquella on la direcció de reflexió és única donat un angle d'incidència, és a dir, que es comporta com un mirall. No obstant, la majoria de materials reals presenten un comportament parcialment difós i especular. Pel cas de materials brillants però no perfectes com un mirall, s'acostuma a definir un lòbul al voltant de la direcció de reflexió especular perfecta.

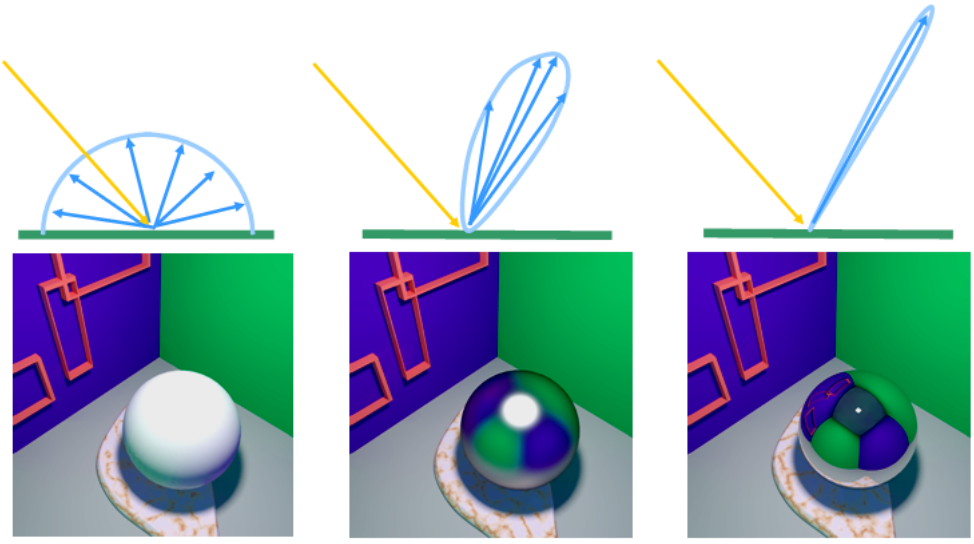

Figura 2.4: Imatge d'un mateix objecte canviant el grau d'especularitat: difós pur, especular imperfecte (*glossy*) i especular pur. *Imatge extreta de [39].*

Per a calcular la direcció de propagació a través de l'objecte quan es produeix una refracció especular, fem servir la **llei de Snell**. Donats dos medis amb índex de refracció η_1 i η_2, l'angle incident θ_1 i l'angle refractat θ_2 la llei de Snell afirma que s'ha de complir:

$$\eta_1 \sin\theta_1 = \eta_2 \sin\theta_2$$

A partir d'aquesta expressió podem calcular l'angle refractat i, per tant, el raig transmès $\vec{T}$ com:

$$\vec{T} = -\frac{\eta_1}{\eta_2}\Psi + \vec{n}_x \left(\frac{\eta_1}{\eta_2}(\vec{n}_x \cdot \Psi) - \sqrt{1 - \left(\frac{\eta_1}{\eta_2}\right)^2 (1 - (\vec{n}_x \cdot \Psi)^2)} \right)$$

Observem també que quan $\eta_1 > \eta_2$, hi ha un angle a partir del qual només tenim *reflexió total interna*. Aquest angle s'anomena angle crític (θ_c) i l'obtenim de: $\sin\theta_c = \frac{\eta_2}{\eta_1}\sin\frac{\pi}{2} = \frac{\eta_2}{\eta_1}$.

Amb les equacions que hem vist, es poden calcular els angles i vectors de reflexió i refracció. Les **equacions de Fresnel** ens permeten, a més, determinar per a una superfície perfectament especular quina part de la llum incident serà reflexada i quina transmesa. Aquestes equacions tenen en compte les dues components de la llum polaritzada:

$$r_p = \frac{\eta_2 \cos\theta_1 - \eta_1 \cos\theta_2}{\eta_2 \cos\theta_1 + \eta_1 \cos\theta_2} \qquad r_s = \frac{\eta_1 \cos\theta_1 - \eta_2 \cos\theta_2}{\eta_1 \cos\theta_1 + \eta_2 \cos\theta_2}$$

Combinant aquestes expressions amb la llei de Snell, podem fer que només dependguin de l'angle incident $\theta_1 = \theta_i$. Per a llum no polaritzada, el valor de la reflectància es calcula com:

$$R_F(\theta_i) = \frac{\|r_p\|^2 + \|r_s\|^2}{2}$$

Observem que es fa servir un mòdul, ja que aquests valors poden ser complexos, com per exemple per als metalls, l'índex de refracció dels quals és un nombre complex. Si assumim que no hi ha ni emissió ni absorció de la llum a la superfície, la transmitància serà $T_F(\theta_i) = 1 - R_F(\theta_i)$.

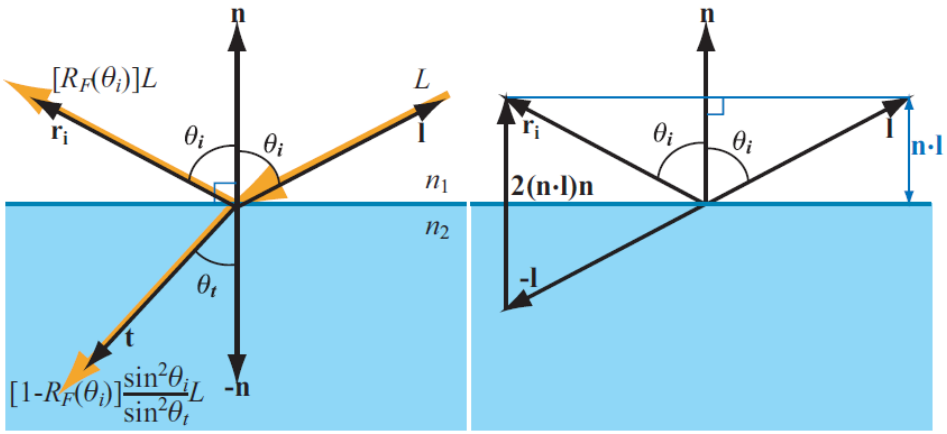

Figura 2.5: Càlcul dels vectors de reflexió i transmissió en superfícies especulars juntament amb el valor de la reflectància. El factor $\frac{\sin^2\theta_i}{\sin^2\theta_t}$ al raig transmès té en compte que amb la refracció la proporció de radiància transmesa i incident és diferent, ja que ha variat l'àrea projectada i l'angle sòlid. *Imatge extreta de [2].*

Com calcular les equacions de Fresnel pot ser costós, normalment es fa servir l'**aproximació de Schlick**, que té la forma:

$$R_F(\theta_i) = R_0 + (1 - R_0)(1 - cos\theta_i)^n$$

on R_0 és la reflectància per la direcció de la normal i, habitualment, $n = 5$.

2.2.4 Models de reflexió

La majoria de materials reals no es poden representar amb una BRDF perfectament difosa o perfectament especular. Sovint, els materials tindran una BRDF bastant complexa. Per a la generació d'imatges per ordinador, s'han desenvolupat diversos models.

El model més senzill i que ja hem vist és el model de **Lambert**, que es fa servir per a materials idealment difosos. En aquest cas, el model és una constant anomenada constant difosa:

$$f_r(x, \Psi \leftrightarrow \Theta) = k_d = \frac{\rho_d}{\pi}$$

Un dels models més populars històricament, ja que es va implementar al hardware de les targes gràfiques, ha estat el model de **Phong**, que ens permet modelar un lòbul de reflexió especular a partir del cosinus de l'angle entre la direcció de sortida i el raig reflectit. La forma d'aquest lòbul es pot controlat amb un paràmetre s, fent que sigui més prim per a valors majors. Aquest lòbul es multiplica per una constant d'especularitat k_s i s'afegeix a la component difosa de Lambert:

$$f_r(x, \Psi \leftrightarrow \Theta) = k_s \frac{(\vec{R} \cdot \Theta)^s}{\vec{n}_x \cdot \Psi} + k_d$$

Una modificació del model anterior, coneguda com model de **Blinn-Phong** corregeix l'efecte de la reflexió especular per a angles gairebé perpendiculars a la normal. El canvi consisteix en fer servir l'angle entre la normal al punt i el vector $\vec{H} = (\Psi + \Theta)/2$. D'aquesta manera, amb angles propers al pla, es formen taques especulars allargades en comptes d'arrodonides. Podem comprovar que és un model mes realista si mirem, per exemple, els reflexes de llums sobre aigua o terra mullat.

$$f_r(x, \Psi \leftrightarrow \Theta) = k_s \frac{(\vec{H} \cdot \vec{n}_x)^s}{\vec{n}_x \cdot \Psi} + k_d$$

Tot i ser molt utilitzat per la seva senzillesa, és un model empíric i, de fet, no compleix amb algunes propietats de les BRDF com la conservació de l'energia i la reciprocitat de Helmhotz. És fàcil veure que per a angles propers a 90 deg, el cosinus del denominador farà que el quocient tendeixi a infinit. Una possible modificació és eliminar aquest denominador.

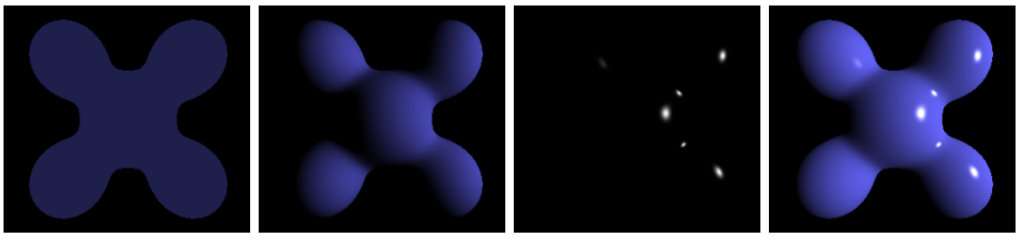

Figura 2.6: Càlcul de la il·luminació d'un objecte fent servir una component constant, la component difosa de Lambert i la component especular de Phong, respectivament. L'última imatge mostra la suma de les tres. *Imatges extretes de la Wikipedia, article "Blinn-Phong shading model".*

Altres exemples de models empírics són el model de **Ward** per a superfícies anisotròpiques o el model de **Lafortune**, que permet recrear les mesures realitzades sobre materials reals modificant i ajustant un conjunt de lòbuls com els de Phong. Pel que fa a models derivats de la física, podem trobar exemples com el model de **Cook-Torrance** o el model **He-Torrance-Sillion-Greenberg**, actualment el més complet però també costós d'avaluar.

També podem trobar-nos amb models de BRDF desenvolupats per a casos concrets. En són exemples el model de **Kajiya-Kay**, per a superfícies extremadament anisotròpiques com per exemple cabells, el model de **Oren-Nayar**, que té en compte la retroreflexió típica de superfícies difoses aspres com l'argila, o el model de **Stam**, que té en compte la difracció de la llum. Com a curiositat, la BRDF de **Hapke/Lommel-Seeliger** modela de manera realista la superfície lunar.

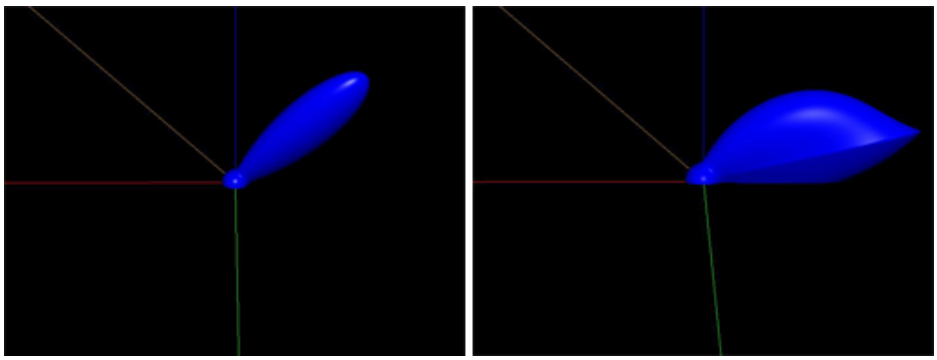

Figura 2.7: Dos exemples de funció BRDF per a una direcció incident donada: a l'esquerra, la BRDF de Blinn-Phong, a la dreta, la de Cook-Torrance. *Imatges obtingudes del programa BRDFLab [15].*

Capítol 3

Generació d'imatges realistes

Al capítol anterior hem vist com es descriuen físicament els fenòmens relacionats amb la propagació de la llum que ens calen per a poder generar una imatge realista d'una escena. Aquest capítol explicarà com es fa per generar aquestes imatges i descriurà els algorismes més destacats.

3.1 L'equació de render

L'objectiu final d'un algorisme d'il·luminació realista és calcular l'estat d'equilibri de la distribució de l'energia de la llum per l'escena. Si assumim que la propagació de la llum és instantània i que ho fa pel buit, aquest equilibri s'obté instantàniament. L'**equació de render** ens descriu aquesta distribució: per a cada punt de superfície x i per a cada direcció Θ obtenim el valor de $L(x \to \Theta)$.

Habitualment, l'equació de render la podrem trobar expressada en l'anomenada **formulació hemisfèrica**:

$$L_o(x \to \Theta) = L_e(x \to \Theta) + \int_{\Omega_x} f_r(x, \Psi \leftrightarrow \Theta) L_i(x \leftarrow \Psi) \cos(\vec{n}_x, \Psi) d\omega_\Psi$$

Si analitzem l'equació, veiem que ens diu que la radiància L_o que es reflecteix en una direcció Θ és la suma de dues components. D'una banda, tenim la radiància emesa L_e, que serà diferent de 0 per a les fonts de llum. L'altra component és la integral sobre tot l'hemisferi centrat a x de les radiàncies incidents L_i des de les possibles direccions Ψ.

Una altra formulació utilitzada és la **formulació per àrea**. Aquesta formulació substitueix l'integral sobre l'hemisferi per una integral sobre les superfícies visibles des del punt x:

$$L_o(x \to \Theta) = L_e(x \to \Theta) + \int_{\Omega_x} f_r(x, \Psi \leftrightarrow \Theta) L_i(y \leftarrow -\Psi) V(x, y) G(x, y) dA_y$$

Aquesta equació té dos termes nous. El primer d'ells és la funció de visibilitat. Si A és el conjunt de totes les superfícies de l'escena, la funció de visibilitat es defineix com:

$$\forall x, y \in A : V(x, y) = \begin{cases} 1 & \text{si } x \text{ i } y \text{ són mútuament visibles} \\ 0 & \text{si } x \text{ i } y \text{ no són mútuament visibles} \end{cases}$$

L'altre terme s'anomena factor geomètric i depèn de la geometria relativa entre les superfícies als punts x i y:

$$G(x, y) = \frac{\cos(\vec{n}_x, \Psi) \cos(\vec{n}_y, \Psi)}{r_{xy}^2}$$

Finalment, una última formulació que ens pot ajudar a acabar d'entendre l'equació és l'anomenada **formulació de llum directa i indirecta**. Si partim de que la radiància sortint de x en direcció Θ és la suma de l'emesa i la que es reflexa, podem expressar-ho també com la suma de la radiància emesa des de superfícies visibles (llum directa) i la que arriba després d'haver rebotat a les superfícies visibles (llum indirecta):

$$\begin{aligned}
L_o(x \to \Theta) &= L_e(x \to \Theta) + L_r(x \to \Theta) \\
L_r(x \to \Theta) &= \int_{\Omega_x} f_r(x, \Psi \leftrightarrow \Theta) L_i(x \leftarrow \Psi) \cos(\vec{n}_x, \Psi) d\omega_\Psi \\
&= L_{\text{directa}} + L_{\text{indirecta}} \\
L_{\text{directa}} &= \int_A f_r(x, \overrightarrow{xy} \leftrightarrow \Theta) L_e(y \to \overrightarrow{yx}) V(x,y) G(x,y) dA_y \\
L_{\text{indirecta}} &= \int_{\Omega_x} f_r(x, \Psi \leftrightarrow \Theta) L_i(x \leftarrow \Psi) cos(\vec{n}_x) i \\
L_i(x \to \Theta) &= L_r(z \to -\Psi) \\
&\quad \text{amb } z = x + t_{min}\Psi, \;\; t_{min} = \{t : t > 0, x + t\Psi \in A\}
\end{aligned}$$

Totes aquestes equacions estan basades en l'equació original que va plantejar Kajiya l'any 1986 [21]. La importància d'aquesta equació va ser que va mostrar que els dos principals grups d'algorismes del moment, els basats en Ray Tracing (secció 3.3.1) i els basats en Radiositat (secció 3.3.2) en realitat estaven solucionant les mateixes equacions basant-se en diferents aproximacions.

3.2 Classificació dels models d'il·luminació

Hem vist que per calcular la radiància a un punt x fem servir l'equació de render, i que aquesta inclou un terme recursiu on intervenen altres punts de l'escena. No obstant, a causa de la complexitat i el cost d'aquest terme també s'han desenvolupat models més senzills que no tenen en compte la resta de l'escena. En funció de si els tenen en compte o no, els classifiquem en models locals o globals.

3.2.1 Models locals

Els models locals són aquells que no tenen en compte la resta d'objectes de l'escena. Per tant, només tenen en consideració la llum directa. Podríem escriure l'equació que fan servir com:

$$L_o(x \to \Theta) = \int_A f_r(x, \overrightarrow{xy} \to \Theta) L_e(y \to \overrightarrow{yx}) G(x,y) dA_y$$

És a dir, que la radiància serà la integral de la radiància emesa per totes les fonts de llum que es trobin sobre l'hemisferi. Fixem-nos que aquesta expressió no és la mateixa que la que havíem vist a l'apartat anterior per a la radiància L_{directa}. La diferència és que aquella tenia en compte el factor de visibilitat per determinar si els dos punts eren mútuament visibles. Per tant, no sabrem si la radiància emesa per una llum és obstruïda abans d'arribar a x, de manera que no tindrem ombres a l'escena.

Sovint, els models locals simplifiquen encara més l'equació i només consideren llums puntuals. Així, si tenim en compte que només considerem la radiància emesa, que aquesta

és diferent de zero a les fonts de llum i que aquestes són un punt, podem expressar l'equació del render com un sumatori

$$L_o(x \to \Theta) = \sum_{i=0}^{N} f_r(x, \overrightarrow{xp_i} \to \Theta) L_{e_i}(\overrightarrow{p_i x})$$

on $L_{e_i}(\varphi)$ és la radiància emesa per la font i, situada a p_i, en direcció φ.

Degut a la seva senzillesa, aquest model va ser adoptat pel hardware gràfic juntament amb la BRDF de Phong. Les targetes gràfiques segueixen un paradigma projectiu de la geometria: els objectes són projectats sobre el pla de la càmera i es calculen quins píxels ocupen abans de calcular el color. Per tant, al moment de pintar un píxel no es té informació sobre la resta de l'escena. Nombroses tècniques s'han proposat per a representar miralls, ombres, llum indirecta, etc. fent servir aquesta rasterització, però queden fora dels objectius i l'abast d'aquest projecte.

3.2.2 Models globals

Al contrari dels models anteriors, els models globals sí que tenen en compte la resta d'objectes de l'escena. Això no vol dir, però, que ofereixin una solució completa a l'equació de render. Com veurem a la secció següent, alguns algorismes se centren en només alguns dels tipus d'interaccions de la llum.

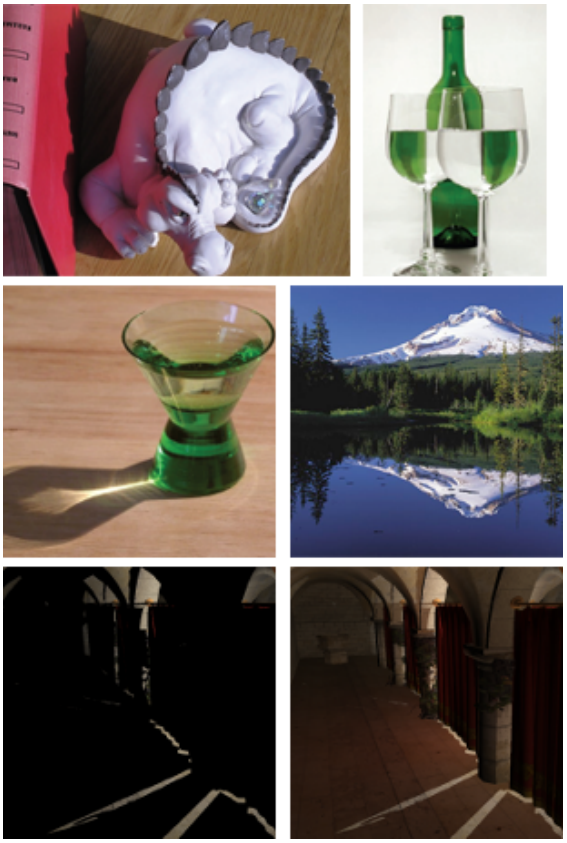

Figura 3.1: Diferents efectes de la il·luminació global. Les 4 primeres imatges mostren fotografies on s'aprecia el *color bleeding*, ombres, refraccions, càustiques i reflexions. Les dues últimes imatges, generades amb un algorisme d'il·luminació global, mostren la diferència entre considerar només la llum directa i tenir en compte les reflexions indirectes. *fotografies i imatges extretes de [23] i [18].*

Per tal de poder classificar millor quines interaccions pot simular un algorisme, es fa servir la **notació del transport de la llum**. Aquesta notació ens dona una representació compacta dels tipus d'interaccions que fa un raig de llum al llarg del seu recorregut fent servir una expressió regular. Els símbols que es fan servir són:

`L`	per a les fonts de llum
`E`	per a l'observador
`D`	per a les reflexions difoses
`S`	per a les reflexions especulars

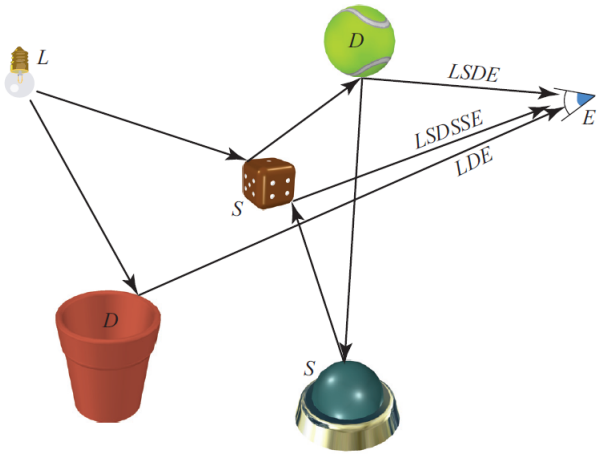

Figura 3.2: Escena amb una llum, un observador, dos objectes difosos i dos d'especulars. Es poden veure diversos camins dels rajos de llum i la seva notació associada. *Imatge extreta de [2].*

Per exemple, `LSDDE` seria la notació d'un raig que començant a la font de llum (`L`) ha fet una reflexió especular (`S`), seguidament dues difoses (`DD`) i ha arribat a l'observador (`E`). S'assumeix que podem separar la BRDF en components difosa i especular. En alguns casos podem trobar-nos amb un terme `G` per a les reflexions imperfectes (de l'anglès, *Glossy*).

Podem també descriure conjunt de possibles camins en forma d'expressió regular. Habitualment es fa servir la notació següent:

`(k)+`	un o més esdeveniments de tipus k
`(k)*`	zero o més esdeveniments de tipus k
`(k)?`	zero o un esdeveniment de tipus k
`(k|k')`	un esdeveniment de tipus k o un de tipus k'

Així doncs, l'expressió `L(S|D)+DE` representa el conjunt de camins iniciats a la llum que tenen una o més reflexions difuses o especulars i finalment arriben a l'ull després d'una última reflexió difosa.

3.3 Algorismes d'il·luminació global

Abans que Kajiya unifiqués amb una mateixa equació els mètodes existents per a il·luminació global, es parlava de dos grans grups d'algorismes. D'una banda, els que es basaven en traçat de rajos recursius: els algorismes de Ray Tracing. Per l'altra banda, els algorismes basats en una discretització de l'escena i el mètode dels elements finits: els algorismes de Radiosity.

3.3.1 Ray Tracing

Els algorismes de Ray Tracing (traçat de rajos) es basen, com el nom suggereix, en traçar rajos per l'escena. L'esquema bàsic consisteix en traçar un raig per a cada píxel, trobar la intersecció més propera d'aquest raig amb l'escena i llançar més rajos recursivament. La propagació recursiva dels rajos necessita alguna condició d'aturada. Normalment es fa servir un percentatge de contribució efectiva del raig mínim o una profunditat màxima de les crides recursives.

Matemàticament, podem veure que aquests algorismes resolen la integral de l'equació de render mitjançant *integració de Monte Carlo*. Es generen N direccions aleatòries Ψ_i sobre l'hemisferi Ω_x, distribuïdes segons una funció de densitat de probabilitat $p(\Psi)$:

$$\begin{aligned} L_o(x \to \Theta) &= L_e(x \to \Theta) + L_r(x \to \Theta) \\ L_r(x \to \Theta) &= \int_{\Omega_x} f_r(x, \Psi \leftrightarrow \Theta) L_i(x \leftarrow \Psi) \cos(\vec{n}_x, \Psi) d\omega_\Psi \\ \langle L_r(x \to \Theta) \rangle &= \frac{1}{N} \sum_{i=1}^{N} \frac{f_r(x, \Psi_i \leftrightarrow \Theta) L_i(x \leftarrow \Psi_i) \cos(\vec{n}_x, \Psi_i)}{p(\Psi_i)} \end{aligned}$$

El primer algorisme de Ray Tracing va ser proposat per Whitted l'any 1980, i es coneix com a **Whitted's Ray Tracing** o Ray Tracing clàssic [44]. L'algorisme comença llançant un raig des de cada píxel (*raig primari*). Quan es troba la intersecció més propera amb l'escena, podem tenir tres casos:

- Si és una superfície especular, es calcula el raig reflectit segons la llei de la reflexió.
- Si és una superfície transparent, es calcula el raig refractat amb la llei de Snell i es transmet a través de l'objecte.
- Si és una superfície difosa, es calcula el valor de la il·luminació directa. Normalment, es tracen rajos cap a les fonts de llum per determinar si aquestes són visibles (*shadow rays*).

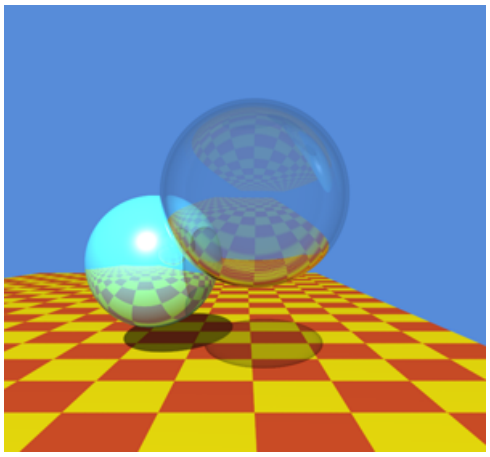

Figura 3.3: Escena clàssica generada amb el Ray Tracing de Whitted.

L'algorisme de Whitted només pot simular els camins de tipus `LD?S*E`, ja que acabem amb la primera superfície difosa que ens trobem des de l'ull, passant per reflexions especulars, o bé incidim a la llum directament des de les reflexions especulars. Per tant, no és

una solució completa a l'equació de render. Fixem-nos també que s'assumeixen reflexions especulars pures.

Cook va presentar una modificació del Ray Tracing de Whitted, anomenada **Distributed Ray Tracing** [9], que permetia simular reflexions especulars o transparents imperfectes. En comptes de llançar un únic raig especular o transmès, se'n distribueixen diversos variant la direcció segons el lòbul especular definit a la BRDF. De manera similar, permetia simular efectes com la *profunditat de camp*, distribuint els rajos segons la forma de la lent, o el *motion blur*, distribuint els rajos en diferents instants de temps pel píxel. No obstant, els camins que és capaç de simular són exactament del mateix tipus que els del seu predecessor.

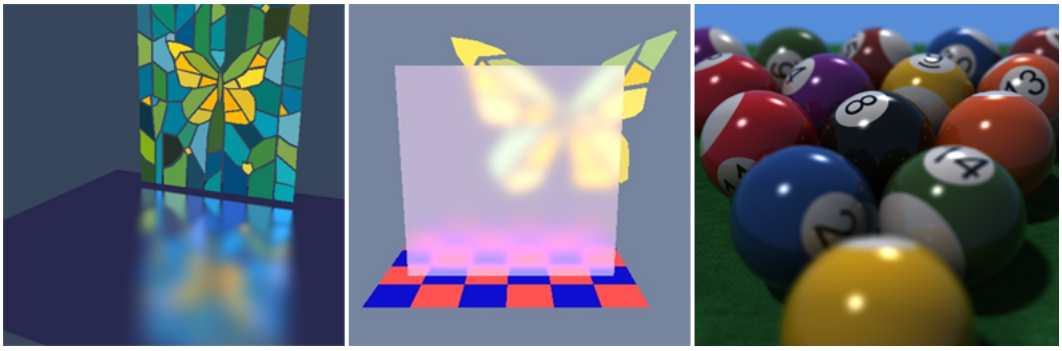

Figura 3.4: Diferents efectes que es poden simular amb el Ray Tracing de Cook. D'esquerra a dreta: reflexions especulars imperfectes, transmissions imperfectes i profunditat de camp.

Kajiya, al mateix article on presentava l'equació de render, va presentar també un algorisme de traçat de rajos que contemplava tots els possibles camins: el **Path Tracing** [21]. Per a cada píxel de la imatge es creen un nombre molt elevat de rajos, que pot arribar a ser de l'ordre de milers. Quan un raig interseca amb una superfície, no es creen més rajos sinó que es decideix aleatòriament i basant-se en la BRDF la direcció per la qual seguirà el raig. Per tant, podem seguir propagant els rajos des de superfícies difoses i modelar superfícies especulars imperfectes, tenint així camins de qualsevol tipus: `L(D|S)*E`. El problema del Path Tracing és que si no llancem prou rajos per píxel, la imatge resultant tindrà molt de soroll.

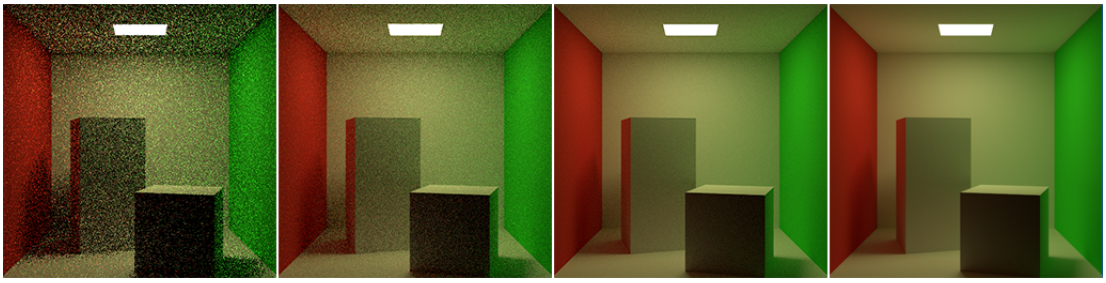

Figura 3.5: Comparació d'una imatge generada amb Path Tracing fent servir 1, 16, 256 i 4096 rajos per píxel.

El cas més general que podem tenir consisteix en aplicar directament l'estimació per Monte Carlo cada vegada que tinguem una interacció amb una superfície. Aquest tipus d'algorismes de Ray Tracing s'acostumen a anomenar **Ray Tracing estocàstic** o **Ray Tracing de Monte Carlo**. És fàcil veure que també contemplen tots els camins `L(D|S)*E`.

Com al cas anterior, podem tenir problemes amb el soroll de la imatge final. Per a millorar els resultats s'apliquen tècniques de reducció de la variança dels estimadors.

Hi ha també variants que tracen els rajos inicials des de la llum en comptes de fer-ho des de l'observador. Són l'algorisme dual dels algorismes de Ray Tracing i se les coneix com algorismes de **Light Tracing** [3]. En funció de l'escena i del tipus de materials, pot ser preferible l'ús d'uns mètodes o dels altres. Per exemple, per a simular càustiques, l'efecte que es produeix quan la llum és focalitzada sobre un punt, els algorismes de Light Tracing requereixen menys rajos que els de Ray Tracing. Els camins `LS+DE` que porten a la formació de càustiques tenen una probabilitat baixa, però la seva contribució visual és important. Amb Light Tracing podem enfocar rajos des de la llum cap a les superfícies especulars directament.

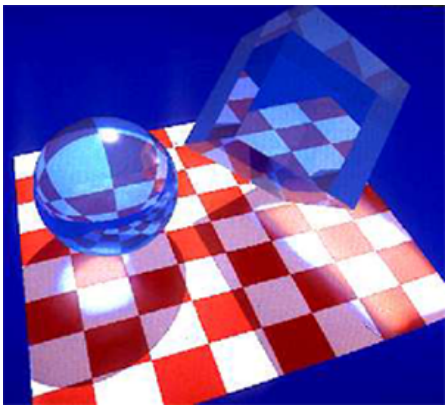

Figura 3.6: Dos objectes de vidre i les càustiques generades amb Light Tracing, *extreta de [3].*

3.3.2 Radiosity

Una altra estratègia que es va plantejar per a solucionar l'equació de render va ser assumir que totes les reflexions són perfectament difoses, és a dir, que segueixen la BRDF de Lambert. A més, en comptes de la radiància, aquests algorismes calculen la radiositat a una malla de l'escena discretitzada en N cel·les.

De la relació entre radiositat i radiància, podem veure que la radiositat mitjana de la cel·la i amb àrea A_i serà:

$$B_i = \frac{1}{A_i} \int_{S_i} \int_{\Omega_x} L(x \to \Theta) \cos(\Theta, \vec{n}_x) d\omega_\Theta dA_x$$

i hem vist anteriorment que:

$$L(x \to \Theta) = L_e(x \to \Theta) + \int_{\Omega_x} f_r(x, \Psi \leftrightarrow \Theta) L_i(x \leftarrow \Psi) \cos(\vec{n}_x, \Psi) d\omega_\Psi$$

Per a superfícies perfectament difoses, la radiància emesa i la BRDF no depenen de la direcció i aquest val $\rho(x)/\pi$:

$$L(x) = L_e(x) + \int_{\Omega_x} \frac{\rho(x)}{\pi} L_i(x \leftarrow \Psi) \cos(\vec{n}_x, \Psi) d\omega_\Psi$$

Si convertim la integral de formulació hemisfèrica a formulació per àrea i fem servir la relació $B(x) = \pi L(x)$ obtenim l'*equació integral de la radiositat*:

$$B(x) = B_e(x) + \rho(x) + \int_S G(x, y) V(x, y) B(y) dA_y$$

Si tornem a prendre la primera equació i apliquem els canvis per a superfícies difoses:

$$B_i = \frac{1}{A_i}\int_{S_i} L(x)\int_{\Omega_x}\cos(\Theta,\vec{n}_x)d\omega_\Theta dA_x = \frac{1}{A_i}\int_{S_i} L(x)\pi dA_x = \frac{1}{A_i}\int_{S_i} B(x)dA_x$$

Discretitzant aquesta última equació i considerant la reflectivitat constant a cada cel·la, arribem al *sistema d'equacions de la radiositat*:

$$B_i = B_{e_i} + \rho_i \sum_j F_{ij}B_j$$

Els factors F_{ij} s'anomenen *factors de forma entre cel·les* i es calculen com:

$$F_{ij} = \frac{1}{A_i}\int_{S_i}\int_{S_j} G(x,y)V(x,y)dA_y dA_x$$

Intuïtivament, podem dir que la radiositat a una cel·la i depèn de dos valors: la pròpia radiància emesa i la fracció de la irradiància (radiositat incident) que es reflecteix. El factor de forma ens indicaria quina fracció de la irradiància a i prové de j. Matemàticament, es pot veure que són algorismes basats en *mètodes d'elements finits*. Aquests mètodes ja s'havien aplicat per a modelar problemes de transferència de calor. La seva primera aplicació a la generació d'imatges va ser l'any 1984 [16], pocs anys després de l'aparició del Ray Tracing clàssic.

Com és d'esperar, la mida de la discretització afectarà a la qualitat de la imatge final però també a la memòria requerida. Tot i que en principi podria semblar que la complexitat de l'algorisme recau a la solució del sistema d'equacions, doncs tenir unes 100 000 cel·les és bastant comú, a la pràctica aquest és bastant estable i és senzill de solucionar amb mètodes iteratius com el de Jacobi o el de Gauss-Seidel. La part realment complicada és el càlcul dels factors de forma, ja que ocupen massa a memòria i a més cal solucionar integrals de dimensió 4. La recerca ha fet especial èmfasi en solucionar aquests problemes.

Els algorismes de radiositat, per tant, simulen camins únicament del tipus `LD*E`, aquells pels quals la llum es propaga en reflexions difoses. Un fenomen que permeten modelar, a diferència del Ray Tracing clàssic, és la propagació del color entre superfícies properes, que es coneix com *color bleeding*.

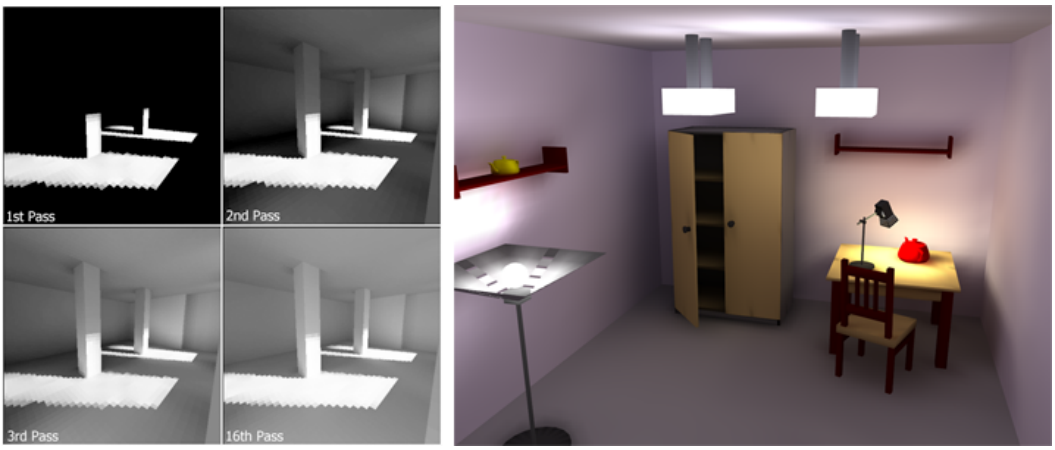

Figura 3.7: A l'esquerra, evolució de la solució del sistema de radiositat després de 1, 2, 3 i 16 iteracions. A la dreta, imatge d'una escena després de 79 iteracions. *Imatges extretes de la Wikipedia, article "Radiosity (3D computer graphics)".*

3.3.3 Algorismes híbrids i multipàs

Hem vist que els algorismes de Ray Tracing funcionen especialment bé per a reflexions especulars. En canvi, els de radiositat tenen com a punt fort les reflexions difoses. És bastant natural, doncs, pensar en possibles combinacions dels dos mètodes i tenir així algorismes híbrids que combinin el millor de cada món. Sovint, són algorismes que requeriran de més d'un pas per a generar la imatge, per això també se'ls anomena algorismes multipàs.

Les primeres tècniques híbrides consistien en fer servir radiositat i afegir els efectes de les reflexions especulars posteriorment amb un pas de Ray Tracing. Es pot afegir fins i tot un pas addicional de Light Tracing per a les càustiques. A continuació, veurem breument les idees d'algunes de les tècniques multipàs més importants.

Final Gathering

Als algorismes de radiositat hem calculat el valor d'aquesta a una malla discretitzada en cel·les de l'escena, i es poden interpolar els valors per a calcular la radiositat a un punt concret. No obstant, si les cel·les no són molt petites es poden perdre alguns detalls importants, com per exemple la forma de les ombres. Una solució mes acurada consisteix en calcular la radiositat a la malla discretitzada en un primer pas i seguidament fer servir Ray Tracing per a generar la imatge. La diferencia amb el Ray Tracing estocàstic és que ara l'avaluació de la integral per a la il·luminació indirecta no serà recursiva, sinó que els rajos que llancem llegiran el valor de la radiositat de la superfície on impactin.

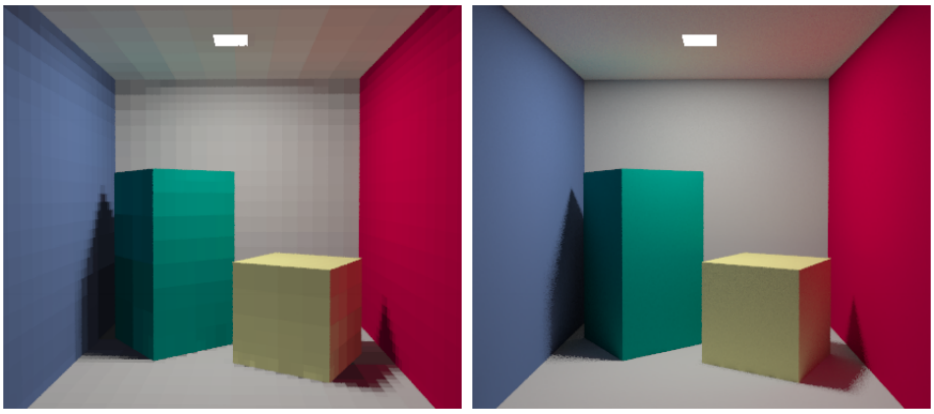

Figura 3.8: Visualització directa de la solució obtinguda per radiositat, a l'esquerra, i imatge generada per Ray Tracing amb Final Gather, a la dreta. *Imatges extretes de [39].*

Bidirectional Path Tracing

La idea principal del Bidirectional Path Tracing [24] és aprofitar que alguns camins és millor generar-los des de l'observador (Path Tracing) mentre que d'altres és millor fer-ho des de la llum (Light Tracing). El camí des de l'observador el denotarem com $E\,x_1\,x_2\ldots x_n$, i el de la llum $L\,y_1\,y_2\ldots y_m$. Aleshores, per cada parell x_i, y_j comprovarem si els dos punts són mútuament visibles i, en cas de ser-ho, afegirem la seva contribució ponderada al resultat del píxel en qüestió.

Els resultats d'aquest algorisme poden tenir soroll, igual que passava amb el Path Tracing, ja que les mostres que generem per cada píxel són un superconjunt de les que genera el Path Tracing. No obstant, degut a la combinació de diversos camins per cada mostra, reduïm abans el soroll amb menys mostres.

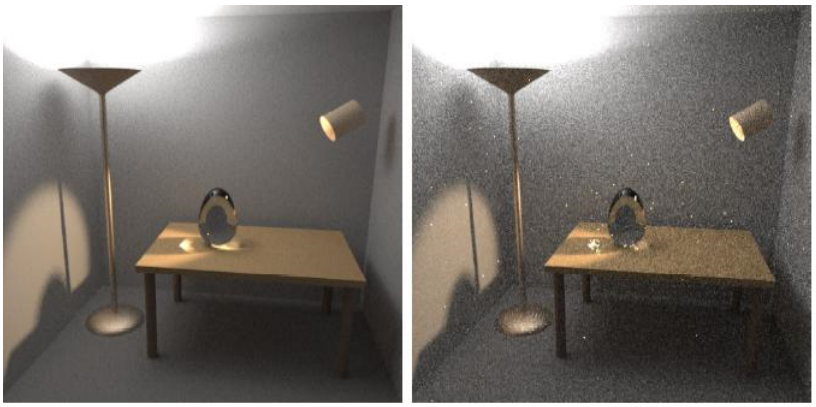

Figura 3.9: La mateixa escena generada amb Bidirectional Path Tracing (esquerra) i Path Tracing normal (dreta). Es pot veure clarament com es redueix el soroll i les càustiques estan més definides.

Metropolis Light Transport

Aquest algorisme [41] es basa en l'adaptació a la generació d'imatges de la tècnica de mostreig de Metropolis i es basa en aprofitar el coneixement de l'espai de mostres de manera més eficient. Inicialment, es crea aleatòriament un conjunt de camins de mostra, per exemple fent servir Bidirectional Path Tracing. Després, l'algorisme se centrarà en produir mutacions d'aquells camins que puguin contribuir més a la imatge final. Aquestes modificacions consisteixen bàsicament en afegir o eliminar vèrtexs del camí i en pertorbar-los per a generar els camins de lents o càustiques.

La idea darrere l'algorisme és que els camins importants poden ser difícils de trobar, però un cop se n'ha trobat un, successives variacions ens portaran a d'altres camins importants. Els seus punts forts són escenes on una regió petita és la responsable de la major part de la il·luminació, com per exemple un forat a la paret o les refraccions que originen les càustiques. Per a escenes molt difoses on hi ha molta contribució indirecta l'assumpció que fa ja no funciona tant bé.

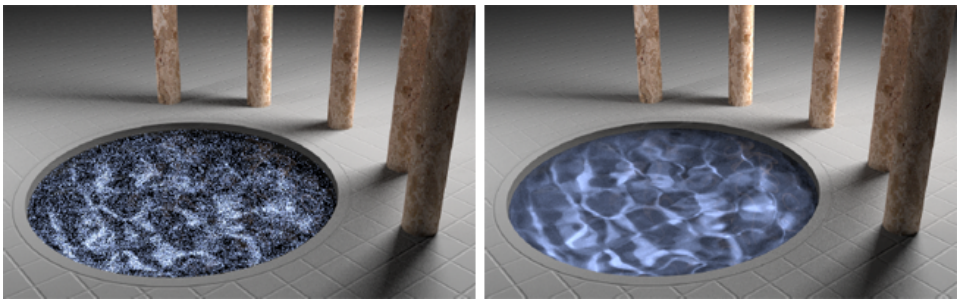

Figura 3.10: La mateixa escena generada en el mateix temps fent servir: Path Tracing Bidireccional amb 210 mostres per píxel (esquerra) i Metropolis Light Transport amb una mitja de 100 mutacions per píxel (dreta).*Imatge extreta de [41].*

Irradiance Caching

Als algorismes de Ray Tracing estocàstic, calcular la il·luminació difosa indirecta a un punt pot requerir traçar molts rajos des d'aquell punt i aquests, al seu torn, poden tornar a generar molt més, fent que l'algorisme sigui extremadament lent. La tècnica de Irradiance Caching [43] aprofita el fet que la irradiància sobre les superfícies de l'escena varia de

manera suau. L'algorisme té una estructura de dades, la *irradiance cache*, on va desant els valors prèviament calculats de la irradiància. Sempre que és possible, els reutilitza interpolant-ne uns quants per a calcular la irradiància de les superfícies properes al punt que estem tractant. En cas de no tenir prou mostres properes, es calcularà traçant rajos la irradiància al punt i aquest valor calculat es guardarà a la *cache*.

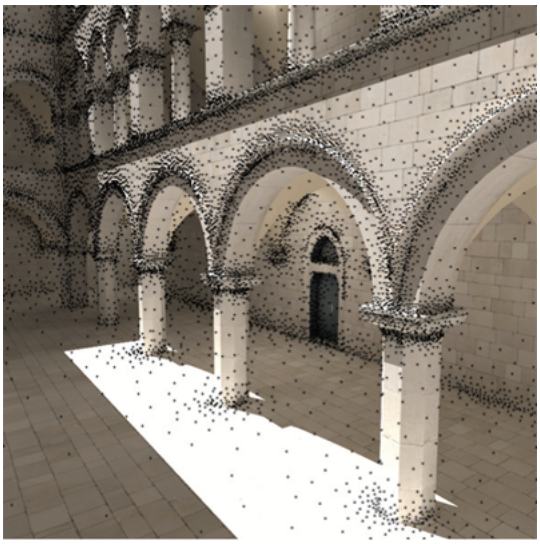

Figura 3.11: Imatge generada amb Irradiance Caching i amb les posicions on la *irradiance cache* ha guardat el valor de la irradiància superposades. Es pot observar com a cantonades i zones corbades té més punts que a les planes.*Imatge extreta de [23].*

Instant Radiosity

L'algorisme de Instant Radiosity [22] està centrat també en accelerar el càlcul de la il·luminació difosa indirecta per a algorismes com el Path Tracing bidireccional. La idea principal és substituir aquest càlcul per un conjunt de llums puntuals virtuals (*virtual point lights*, VPL). Inicialment, es traça un conjunt de camins des de la llum i es detecten aquelles zones de les superfícies on hi incideixin més trajectòries, col·locant llums virtuals. En un segon pas, des de l'observador, es calculen els camins de tipus `LDS*E`. Quan s'arriba a una superfície difosa, s'aplica il·luminació directa des de totes les VPL visibles pel punt de la superfície. L'autor de l'algorisme també proposa fer servir el hardware gràfic per a calcular la il·luminació amb les VPL en comptes de fer el segon pas amb Ray Tracing.

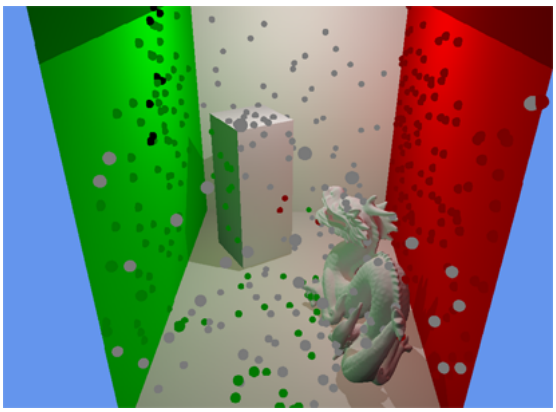

Figura 3.12: Representació de les llums virtuals que es faran servir per a il·luminar l'escena. *Imatge extreta de [18].*

Capítol 4

Photon Mapping

Al final del capítol anterior hem vist algorismes híbrids i multipas amb els quals es podien obtenir imatges de millor qualitat i sovint en menys temps que fent servir els mètodes senzills. Un altre mètode multipàs que no hem comentat, ja que serà l'objectiu de tot aquest capítol analitzar-lo detalladament, és el **Photon Mapping** proposat per Henrik Wann Jensen l'any 1996 [19, 20].

De manera resumida, l'algorisme té dos passos. El primer pas consisteix en llançar fotons des de les fonts de llum i guardar-los en una estructura de dades adient. El segon pas, que generarà la imatge de l'escena, consultarà aquesta estructura de dades per a consultar els fotons propers i així estimar la il·luminació

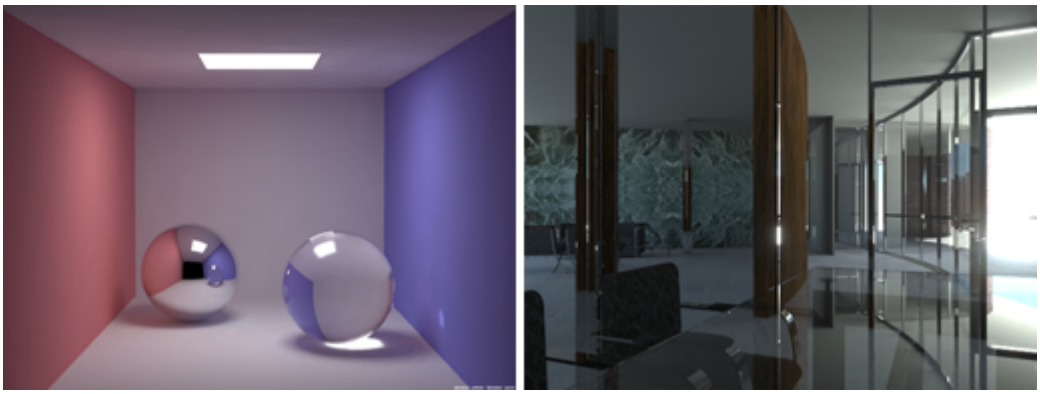

Figura 4.1: Imatges d'exemple generades amb Photon Mapping, extretes de [20].

4.1 Primer pas: traçat de fotons

De manera semblant al funcionament bàsic dels algorismes de Light Tracing, el primer pas de l'algorisme de Photon Mapping consisteix en traçar fotons des de les fonts de llum i fer que es propaguin per l'escena.

Els fotons transportaran el flux des de les fonts de llum. Si des d'una llum s'emetran N fotons, cadascun d'ells portarà associat un flux de P/N, on P és la potència total de la font de llum. Els tipus de llums que suporta l'algorisme són arbitraris: llums puntuals, amb àrea, direccionals... Només cal saber com emetre els fotons correctament. Per a les llums puntuals, es tria una direcció aleatòriament. Per a les llums amb àrea, per exemple un plafó quadrat al sostre o una esfera lluminosa, normalment se selecciona una posició sobre la superfície i després una direcció. Per a les llums direccionals, una opció seria

projectar el volum englobant de l'escena en la direcció de la llum i seleccionar una posició dins de l'àrea projectada. Per a llums anisotròpiques, que no emetin la mateixa potència en totes direccions, un cop decidits el punt d'origen i la direcció podem escalar el valor del flux associat al fotó segons la definició de la llum. Per a fer el flux de tots els fotons el més semblant possible quan tenim diverses fonts de llum alhora, s'acostuma a emetre més fotons des d'aquelles fonts més brillants.

Un cop emesos, els fotons es propagaran per l'escena. Quan un fotó interseca un objecte, poden passar tres coses: es reflecteix, es transmet o s'absorbeix. La decisió sobre quina de les tres aplicar es farà a partir de les probabilitats expressades per la BRDF. El flux del fotó es veurà afectat en aquestes successives interaccions. Per decidir quan cal deixar de propagar el fotó es pot fixar un nombre màxim de reflexions o fer servir una probabilitat d'absorció inversament proporcional al flux que transporta, de manera que cada vegada sigui més probable acabar el seu recorregut.

A vegades, quan s'emeten els fotons, ens pot interessat fer especial èmfasi sobre algunes regions de la imatge, per exemple sobre objectes especular o transparents que poden donar lloc a càustiques. Si llancem fotons cap a tot arreu, pot ser que hi arribin pocs sobre aquestes superfícies i calgui d'elevar molt el nombre total de fotons emesos per a visualitzar posteriorment la càustica. Per tant, a part de fer una emissió general de fotons, podem fer-ne d'altres focalitzades a les regions d'interès.

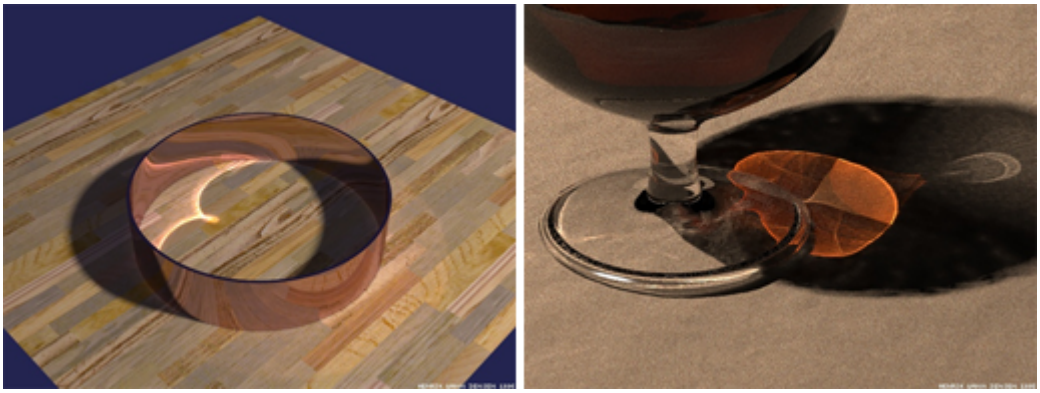

Figura 4.2: Exemples de càustiques generades amb Photon Mapping per a un anell i una copa de licor, extrets de [20]. Photon Mapping és un algorisme molt bo per a simular aquest fenomen a les imatges que generem, però pot fer falta focalitzar els fotons cap a les superfícies que donaran lloc a aquestes càustiques.

4.2 El mapa de fotons

El mapa de fotons o *Photon Map*, d'aquí el nom de l'algorisme, és l'estructura de dades que farem servir per a emmagatzemar els fotons. Sempre que un fotó impacti sobre una superfície difosa, ens voldrem guardar informació sobre aquesta interacció. Fixem-nos que no cal fer-ho per a les reflexions especulars, doncs la probabilitat que els fotons impactin amb la direcció que després veurem aquella superfície es pot considerar nul·la. Per a generar reflexions especulars precises a la imatge final, veurem que es fa servir Ray Tracing.

Les dades que típicament ens interessarà saber sobre els fotons seran: la **posició** on ha impactat, la **direcció** d'incidència i el **flux**. La posició farà falta per poder tenir la representació del mapa de fotons separada de la de l'escena. El flux, evidentment, serà la magnitud física que ens permetrà estimar la radiància a un punt. La direcció servirà

per a calcular el valor de la BRDF quan fem la visualització i per a distingir si els fotons realment són propers a una superfície. Imaginem una paret molt prima, a la qual han incidit fotons pels dos costats. Quan visualitzem un costat de la paret, no volem que els de l'altre contribueixin a la radiància. Saber la direcció d'impacte ens permetrà tractar aquests casos.

Un altre factor a considerar és fer servir més d'un *Photon Map*. L'autor proposava, de fet, fer-ne servir dos, que anomenava *global photon map* i *caustics photon map*. El segon el feia servir per a emmagatzemar aquells fotons focalitzats sobre les superfícies especulars, i era més útil per a generar les càustiques. El primer contenia la resta de fotons emesos de manera normal, i representa la il·luminació general de l'escena. Com veurem a la següent secció, és útil tenir aquests dos mapes de fotons per separat per a augmentar la qualitat de la imatge final.

El mapa de fotons el farem servir durant el segon pas, quan visualitzem l'escena. Sabem que serà consultat de manera estàtica, ja que ja no se'n generaran de nous. Veurem també tot seguit que ens interessarà cercar-hi els n fotons més propers a un punt donat. Una estructura de dades adient per a informació espacial estàtica i consultes de proximitat eficients és el *kd-tree*. Un *kd-tree* és una estructura de particionat de l'espai k-dimensional. Per a fer les particions, es fan servir únicament plans perpendiculars als eixos de coordenades.

La construcció d'un *kd-tree* és molt senzilla. Es comença a partir d'un punt seguint un criteri de selecció, per exemple el que correspongui al valor de la mediana en alguna de les dimensions, i se separen la resta dels punts segons si la seva coordenada corresponent és major o menor a la seleccionada. Recursivament, tornem a aplicar aquest procediment a cadascun dels dos grups que s'han creat. L'elecció de la coordenada segons la qual particionem també pot seguir diversos criteris, però sovint es va triant la següent coordenada de manera cíclica (per exemple, en tres dimensions: $X\,Y\,Z\,X\,\ldots$). Altres criteris més elaborats, per exemple, són buscar els punts a major distància entre ells en alguna de les coordenades o la coordenada amb variància major.

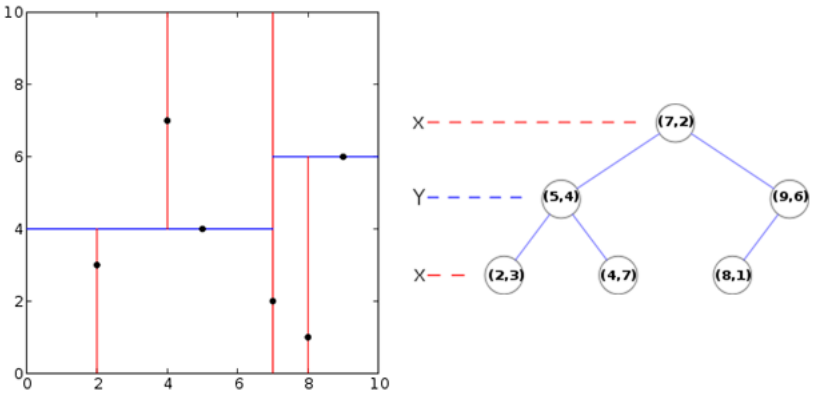

Figura 4.3: Exemple de *kd-tree*. A l'esquerra, es mostra un conjunt de punts i com es va particionant l'espai. A la dreta, l'estructura de dades en arbre generada. *Imatges extretes de la Wikipedia, article "kd-tree".*

La figura 4.3 mostra un exemple de *kd-tree* per a un conjunt de punts en dues dimensions. El criteri de selecció del punt mig ha estat el de la mediana, i per a la dimensió s'ha fet servir el criteri d'alternar entre les dues. Fixem-nos que els nodes intermitjos també

contenen informació de punts, a diferència d'altres estructures de particionat on els punts sempre són a les fulles. Per a representar el *kd-tree*, si aquest és balancejat, podem fer servir un vector: el punt a la posició i tindrà el fill esquerre a la posició $2i$ i el dret a la $2i+1$.

Per a cercar els n fotons més propers a un punt x que són dins d'un radi r, l'algorisme és molt senzill i el podem veure a la figura 4.4. Observem que no cal fer l'arrel als càlculs de les distàncies entre el fotó i el punt x si treballem amb el radi al quadrat.

```
cerca_fotons(p, x, r^2) {
    δ = distància al pla corresponent al node p;
    if (δ < 0){
        baixem pel fill esquerre primer
        cerca_fotons(2p, x, r^2);
        if (δ^2 < r^2)
            cerca_fotons(2p + 1, x, r^2);
    } else {
        baixem pel fill dret primer
        cerca_fotons(2p + 1, x, r^2);
        if (δ^2 < r^2)
            cerca_fotons(2p, x, r^2);
    }
    d^2 = distància al quadrat entre el fotó a p i x;
    if (d^2 < r^2)
        inserim el fotó a la llista de fotons propers;
}
```

Figura 4.4: Algorisme per a cercar els fotons a distància inferior a r respecte un punt x al *Photon Map*.

4.3 Segon pas: visualització de l'escena

Després d'haver traçat els fotons, farem un pas de Ray Tracing per a generar la imatge final de l'escena. Els fotons que hem traçat al primer pas ens permetran calcular la il·luminació global de manera eficient.

El *photon map* representa el flux incident a cada punt de l'escena. Per tant, la densitat de fotons a un punt x és una estimació del valor de la irradiància al punt. La radiància reflectida en una direcció Θ la podem calcular multiplicant la irradiància per la funció BRDF. Per a calcular la densitat de fotons a un punt busquem al *photon map* els N fotons més propers, els quals podrem englobar dins d'una esfera de radi r. Dividint el flux acumulat dels fotons per la projecció de l'esfera sobre la superfície (un cercle d'àrea πr^2) obtenim la densitat de fotons o irradiància. Per tant, la radiància es calcula com:

$$L(x \to \Theta) = \sum_{i=1}^{N} f_r(x, \Theta \leftrightarrow \Psi_i) \frac{\Delta\Phi_i(x \leftarrow \Psi_i)}{\pi r^2}$$

Tot i que podem fer servir el càlcul anterior per a mostrar el valor de la radiància a cada punt visible de la imatge, els resultats que obtindrem seran bastant borrosos i la imatge tindrà poca qualitat si no fem servir un nombre extremadament gran de fotons.

Per això, donada la integral de la radiància reflectida, la podem separar en quatre integrals diferents que s'avaluaran de manera independent. Per a fer-ho, prendrem per separat la component difosa i especular del BRDF, que les denotarem com $f_r = f_{r,D} + f_{r,S}$, i les diferents components de la radiància incident $L_i = L_{i,l} + L_{i,c} + L_{i,d}$, tals que:

- $L_{i,l}$ és la il·luminació directa rebuda des de les fonts de llum visibles pel punt.
- $L_{i,c}$ és la il·luminació indirecta que arriba a través de reflexions o transmissions especulars.
- $L_{i,d}$ és la il·luminació indirecta que arriba després de reflectir-se de manera difosa almenys un cop.

Amb aquests canvis, la integral de la radiància incident l'expressem com:

$$
\begin{aligned}
L_r(x \to \Theta) &= \int_{\Omega_x} f_r(x, \Psi \leftrightarrow \Theta) L_i(x \leftarrow \Psi) \cos(\vec{n}_x, \Psi) d\omega_\Psi \\
&= \int_{\Omega_x} f_r L_{i,l} \cos(\vec{n}_x, \Psi) d\omega_\Psi + && \text{directa} \\
&\quad \int_{\Omega_x} f_{r,S}(L_{i,c} + L_{i,d}) \cos(\vec{n}_x, \Psi) d\omega_\Psi + && \text{especular} \\
&\quad \int_{\Omega_x} f_{r,D} L_{i,c} \cos(\vec{n}_x, \Psi) d\omega_\Psi + && \text{càustiques} \\
&\quad \int_{\Omega_x} f_{r,D} L_{i,d} \cos(\vec{n}_x, \Psi) d\omega_\Psi && \text{indirecta suau}
\end{aligned}
$$

Figura 4.5: A la imatge de l'esquerra, s'ha visualitzat directament la radiància obtinguda del Photon Map. A la de la dreta, s'ha fet servir només per a la il·luminació indirecta al mostrejar l'hemisferi de cada punt visible. *Imatge extreta de [23].*

Per tant, l'algorisme per a la visualització fa servir els dos *Photon Map* com hem comentat abans i tindria les diferents parts:

- La il·luminacio directa per a les superfícies visibles es calcula fent servir Ray Tracing, fent-lo estocàstic si volem contemplar llums amb àrea.
- Les reflexions i transmissions especulars també es fan amb Ray Tracing.
- Les càustiques s'obtenen amb el *caustics photon map*, que permet millor resolució i els fotons ja s'han focalitzat a les superfícies especulars.

- La resta d'il·luminació indirecta es calcula mostrejant l'hemisferi. Per a la radiància incident des del punt y més proper per cada direcció de mostreig, fem servir el *photon map* i la calculem buscant els N fotons més propers a y, com hem vist abans.

Aquesta indirecció addicional, que en certa manera es basa en els algorismes de Final Gathering, aconsegueix reduir les imperfeccions de la imatge final com podem veure a la figura 4.5.

L'algorisme de Photon Mapping també pot ser adaptat per a simular altres fenòmens més complexos. Per exemple, a [20] l'autor explicava com simular medis participatius, *subsurface scattering* o fenòmens de difracció.

Figura 4.6: Altres efectes que es poden simular adaptant l'algorisme: medis participatius (esquerra), *subsurface scattering* (centre) i difracció (dreta). *Imatges extretes de [20].*

4.4 Algorisme progressiu

Degut a la naturalesa de l'algorisme original, és també relativament senzill fer-ne una versió progressiva, és a dir, que no ens faci falta traçar tots els fotons de cop al principi. D'aquesta manera, serà possible aconseguir reduir el temps de càlcul necessari per a generar la imatge i podem aconseguir Photon Mapping a *framerates* interactius.

Aquest algorisme es basa en dues observacions. Per una banda, si no movem la càmera els rajos primaris que tracem al segon pas de Ray Tracing aniran a parar al mateix lloc. D'altra banda, si assumim que quan ens estem movent no ens fa falta una qualitat tant gran a la imatge, podem no traçar tots els fotons de cop i anar traçant-ne més a cada frame, de manera que passats uns instants tindrem ja una imatge amb bona qualitat. L'algorisme es divideix en tres etapes:

- **Pas de càmera**: és un pas de Ray Tracing semblant al de Whitted. La principal diferència és que *només l'executarem si la càmera s'ha mogut.* Els rajos els reflectirem a les superfícies especulars i els transmetrem pels objectes transparents. En comptes de calcular el color de píxels, cada raig primari omplirà una estructura amb les dades següents:
  - La **posició** on ha intersecat el raig.
  - La **normal** de la superfície.
  - La **BRDF** per a la direcció incident.
  - L'**atenuació** de la llum directa acumulada, per a implementar penombres traçant un sol *shadow ray* per iteració.

  - El **radi** r^2 pel qual buscar els fotons propers. Hi posarem el valor fixat per defecte.
  - El **nombre de fotons** N que hem agrupat, de moment 0.
  - El **flux** Φ acumulat, que també serà 0 de moment.

- **Pas de fotons**: aquest pas no varia respecte l'algorisme original. Tracem els fotons des de la font de llum i construïm el *Photon Map*. No obstant, ara executarem aquest pas un cop a cada frame. Els fotons no es reutilitzen mai entre dos frames.

- **Pas de *gather***: aquest pas és la gran diferencia respecte l'algorisme original. Quan hem fet el pas de càmera, hem guardat tota la informació necessària per a poder saber on ha anat a parar el raig primari i les dades del material de la superfície. Com tenim la posició i la normal del punt on hem impactat, podrem buscar al *Photon Map* quins fotons són a prop (dins del radi r^2). També tenim la BRDF necessària per a calcular la radiància a partir del flux que anirem acumulant dels fotons, juntament amb la atenuació que ha patit el raig fins arribar al punt.

Per què cal tenir a cada punt un radi, el nombre de fotons i el flux total? A mesura que anem acumulant més fotons en successius passos de *gather*, el radi de l'estimador hauria de ser més petit per a no incloure tants al següent pas. Es calcula el factor de reducció següent:

$$\rho = \frac{N_{\text{prev}} + \alpha N_{\text{nous}}}{N_{\text{prev}} + N_{\text{nous}}}$$

on N_{prev} és el nombre de fotons prèviament acumulats, N_{nous} els que han caigut dins del radi al frame actual i α és un paràmetre de control que fixarem entre 0 i 1. Amb aquest factor de reducció calcularem:

$$\Phi = \rho\,(\Phi_{\text{prev}} + \Phi_{\text{nou}}) \qquad\qquad r^2 = \rho\, r^2_{\text{prev}}$$

I amb aquests paràmetres nous calcularem l'estimació de la radiància. A la següent iteració, el radi de cerca de fotons ja serà menor perquè no ens farà falta agafar-ne tants. L'efecte d'aquesta reducció és que tindrem més ben definits els contorns de les càustiques. Si, per exemple, deixem sempre el mateix radi, obtindrem càustiques borroses. Si, en canvi, fem decrèixer el radi ràpidament, veurem cada vegada els impactes dels fotons més definits, ja que no hi haurà prou àrea per a agafar-ne diversos i fer una estimació. Per a cada escena caldrà trobar l'equilibri del factor α que ens dóna una bona definició.

La figura 4.7 mostra un exemple d'escena visualitzada fent servir l'algorisme de Photon Mapping Progressiu. A cada iteració s'emeten 65 536 fotons nous. Veiem que tot i que per una sola iteració es noten molt els fotons, ràpidament la imatge se suavitza i en poques iteracions tenim una bona qualitat. La contribució dels nous fotons cada vegada és menor, com podem veure comparant les dues últimes imatges, fet que lliga amb el factor de reducció que acabem de veure. Si no el tinguèssim, la imatge cada vegada estaria més saturada.

La implementació d'aquesta escena d'exemple la podem trobar al *SDK* d'OptiX, un Raytracer interactiu desenvolupat per NVIDIA del qual en parlarem al següent capítol. s'ha executat a un PC amb processador Intel Core 2 Duo E8500 a 3.16GHz, 8GB de memòria RAM i una gràfica NVIDIA GeForce GTX 280 amb 1GB de memòria. S'obté un *framerate* bastant estable de 32 fps si no ens movem, i entre 19 i 22 fps quan no parem de moure la càmera. Per tant, podem considerar que s'executa a temps real.

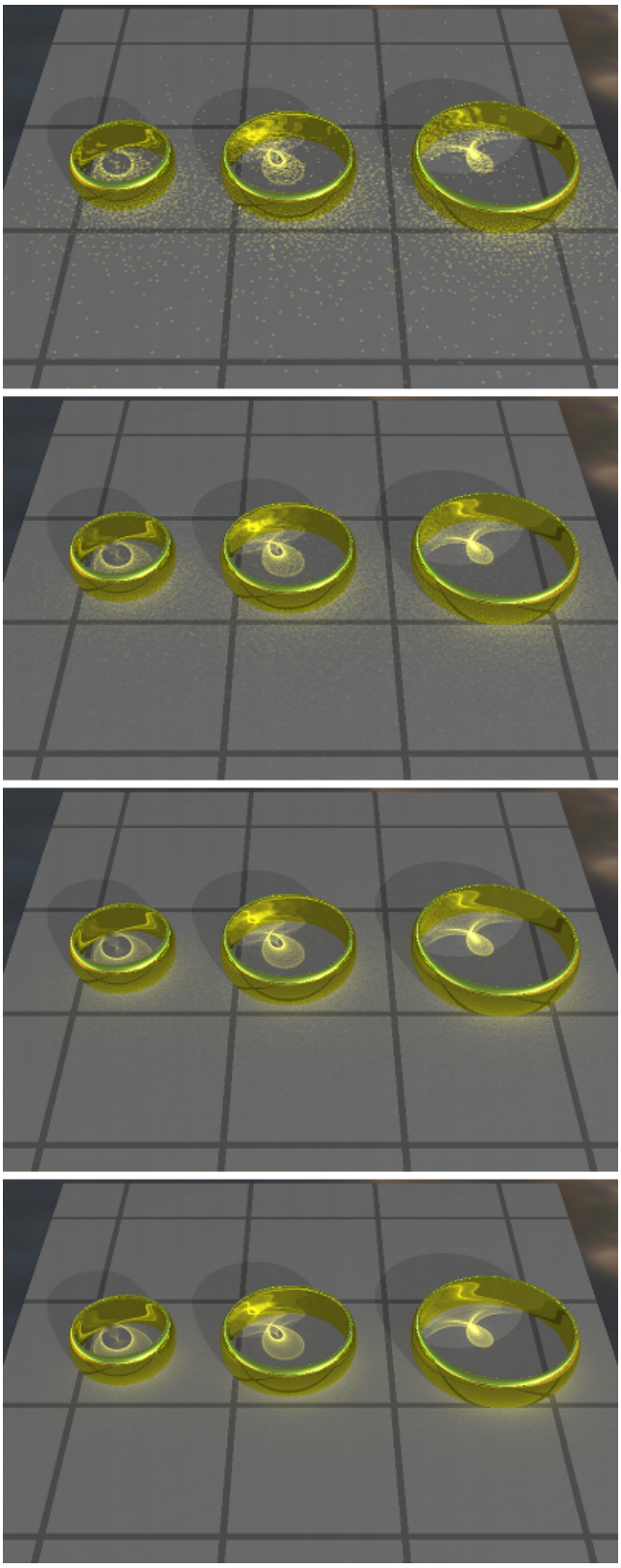

Figura 4.7: Imatge generada amb el Photon Mapping Progressiu en 1, 8, 32 i 128 iteracions.

Capítol 5

Raytracers actuals

Anomenem Raytracers (traçadors de rajos) els programes que es basen en el renderitzat per traçat de rajos. Ja hem vist que són algorismes d'il·luminació global, més costosos que els basats en la rasterització. No obstant, també són molt paral·lelitzables, ja que cada píxel pot ser calculat de manera independent. Gràcies a això, els Raytracers han anat evolucionant i avui en dia s'aconsegueixen rendiments molt bons. Durant aquest capítol, veurem breument l'evolució dels Raytracers interactius i analitzarem les millors solucions actuals.

5.1 Evolució del Ray Tracing interactiu

Durant els anys 90, es van desenvolupar Raytracers que permetien l'execució a temps real en supercomputadors, com per exemple el de Parker et al. [29]. Al 2001, Wald et al. [42] presentaven un Raytracer interactiu que feia ús de les instruccions SIMD[1] per a un clúster de PCs.

Amb l'arribada de les GPU programables, un dels primers passos cap al Raytracing en GPU va ser l'anomenat *Ray engine* [7], el qual realitzava el test d'intersecció raig-triangle fent servir el *vertex shader* i el *fragment shader*. Les aplicacions no es limitaven a Ray Tracing, sinó que es podia fer servir per a altres algorismes com *path tracing*, *photon mapping*, càlcul dels *form factors* per a algorismes de radiositat o algorismes de visibilitat en general. Purcell et al. demostraven com implementar completament algorismes com Ray Tracing [33] i Photon Mapping [34] a la GPU. Inicialment, aquests algorismes feien servir una malla regular per accelerar el càlcul a causa de la dificultat d'implementar estructures més eficients a la GPU. Posteriorment, s'han publicat diversos articles describint recorreguts eficients per altres estructures com els Kd-Tree [14] o les jerarquies de volums englobants (BVH) [17].

Més recentment, s'ha aconseguit fer Raytracers a temps real fent servir CUDA. Per exemple, Shih et al. [36] mostraven renders d'escenes d'uns 2 milions de polígons a una resolució de 1024×1024 amb *framerates* entre 30 i 43 fps. Aila i Laine [1], optimitzant a mà el codi assemblador produït pel compilador de CUDA, aconsegueixen elevar aquesta xifra i situar-la entre 95 i 180 fps.

Hi ha hagut altres aproximacions al Ray Tracing interactiu basant-se en les noves arquitectures, com per exemple la del processador CELL [4], o en tenir un hardware de propòsit específic per a fer Ray Tracing, com l'anomenat Ray Processing Unit (RPU) [45].

[1] Abreviatura de Single Instruction, Multiple Data: una mateixa instrucció s'aplica a la vegada a un conjunt dades, explotant així el paral·lelisme a nivell de dades.

Paral·lelament a aquests avenços, diferents grups han anat creant Raytracers interactius o a temps real que permeten ser programats en certa mesura. Un dels primers va ser el **Star-Ray** [30]. Oferia certa flexibilitat i framerates interactius amb clústers d'ordinadors. Dietrich et al. [10] van proposar l'API d'**OpenRT**, fent servir una interfície similar a OpenGL per a definir càmeres, llums, materials i un bucle de render personalitzat. El codi descrit era optimitzat mitjançant instruccions SIMD. Altres solucions més noves com **Manta** [5], **Razor** [11] o **Arauna** [6] aconsegueixen bons rendiments, però estan concebudes com una solució completa de rendering específic, i es fa difícil integrar-les amb altres sistemes o aplicacions.

D'entre les solucions més actuals, n'hem escollit dues per a analitzar-les a fons als següents apartats: **OptiX** i **OpenRL**.

5.2 OptiX

OptiX és un motor de Ray Tracing programable i de propòsit general desenvolupat per NVIDIA[2]. Les principals característiques són [31]:

- És un motor de Ray Tracing general i de baix nivell: el motor se centra en els càlculs del traçat de rajos, sense entrar en detalls específics de *render*. D'aquesta manera, altres algorismes no gràfics que requereixin traçar rajos, com ara propagació de so, detecció de col·lisions o cerques de camins, es poden programar fàcilment amb OptiX.
- El *pipeline* és programable: podem definir el comportament de certes operacions involucrades al càlcul del traçat de rajos, així com assignar quines dades volem que acompanyin a cada raig.
- Segueix un model de programació simple, ja que el motor abstrau l'agrupament o reordenament de rajos, detalls específics de l'arquitectura o construccions de tipus SIMD (una sola instrucció aplicada a moltes dades).
- Té una representació eficient de l'escena: implementa un model d'objectes que ofereix herència dinàmica per a facilitar la compactació dels paràmetres de l'escena. Aquesta es representa amb un graf que suporta instanciació, diferents nivells de detall i estructures d'acceleració jeràrquiques.
- Interacció tant amb OpenGL com amb DirectX.

Tot i estar basat en CUDA, el seu caràcter de traçador de rajos de propòsit general fa que sigui unes 3 o 4 vegades més lent que un Raytracer específic i optimitzat al màxim [26], com per exemple el que hem comentat abans d'Aila i Laine [1]. A canvi, però, obtenim igualment bons rendiments i ens podem abstraure dels detalls de baix nivell.

5.2.1 Tipus de programes i ordre d'execució

Per a utilitzar OptiX, la nostra aplicació haurà de crear un context d'OptiX des del *host*, per exemple durant la inicialització del nostre programa. A través d'aquest context és com seran enviades les dades al *device*, la GPU. Per a fer una aplicació en OptiX, cal definir un conjunt de programes que han d'estar fets en CUDA C i compilats en format PTX per a passar-li al context. Els diferents tipus de programa que podem crear, mantenint la nomenclatura que es fa servir a la documentació [28], són:

[2] http://www.nvidia.com

Figura 5.1: Exemples d'aplicacions realitzades amb OptiX [31].

- **Ray generation**: és l'encarregat de crear i llançar els rajos primaris i desar els resultats al buffer de sortida. S'executa un cop per cada posició del buffer. Ens permet implementar diferents models de càmera, *super-sampling*, filtrats, mapes de fotons, ... Es poden definir més d'un al context, associant-los un índex diferent. Quan comencem el traçat de rajos, especifiquem quin índex volem fer servir per a cridar el programa, i així podem implementar tècniques que requereixin diverses passades.

- **Intersection**: implementa el test d'intersecció entre el raig i la geometria. Ha de determinar si hi ha intersecció i en quin punt i, si s'escau, calcular dades basades en aquesta posició com ara normals o coordenades de textura interpolades. Gràcies a aquest programa, podem implementar superfícies arbitràries i procedurals, com ara esferes o fractals.

- **Bounding Box**: calcula la capsa englobant d'una primitiva. És invocat durant la fase de construcció de l'estructura d'acceleració de l'escena.

- **Closest Hit**: s'invoca quan s'ha determinat quina és la intersecció més propera d'un raig amb la geometria de l'escena. Típicament, en aquest programa és on calculem la il·luminació i el material d'un objecte. Sovint ens pot interessar llançar rajos addicionals des del punt d'intersecció.

- **Any Hit**: a diferència de l'anterior, aquest programa s'invoca cada vegada que es troba una intersecció entre el raig i l'escena. Per tant, les invocacions no tenen per què venir ordenades per proximitat. Pot decidir si talla el recorregut del raig (per exemple, quan fem tests d'oclusió per a *shadow rays* o *ambient occlusion* o ignorar la intersecció permetent al raig continuar normalment (per exemple, per a implementar *alpha masking*, transparència basada en el canal alpha d'una textura).

- **Miss**: serà el programa invocat quan el raig no intersequi amb la geometria de l'escena. El farem servir per implementar el color de fons, per exemple, amb *environment mapping*.

- **Selector visit**: ens permet escollir per quin dels diversos nodes d'un element de la jerarquia d'escena volem propagar el raig. Habitualment, farem que cada node fill tingui un nivell de detall diferent i, basant-nos en les dades emmagatzemades al raig, seleccionarem per quin avançar.

- **Exception**: ens permet implementar el mecanisme de captura d'excepcions a la GPU. Tenim algunes definides per OptiX, però se'n poden afegir de personalitzades.

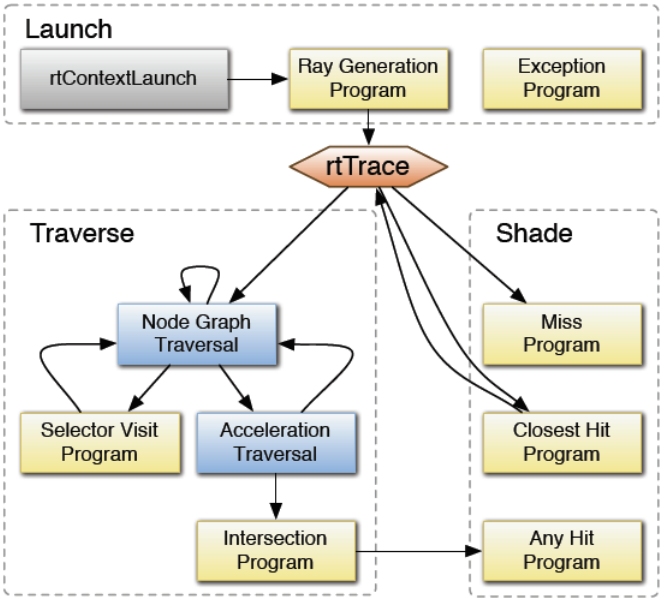

Figura 5.2: Graf de crides entre els programes d'Optix, extret de [31].

La figura 5.2 mostra el graf de crides entre els diferents programes que té OptiX. Quan s'indica al context de començar l'execució, es crida al programa de generació de rajos. L'operació central, anomenada a la figura *rtTrace*, alternarà entre la cerca d'interseccions raig-escena (*Traverse*) i respondre a aquestes interseccions (*Shade*). Per trobar la intersecció, es recorre el graf d'escena, els programes selectors de nodes i les estructures d'acceleració, invocant el programa de test d'intersecció per a obtenir la resposta final del test. Si hi ha intersecció, s'invocarà el programa *Any hit*. Quan se sap quina és la intersecció més propera, es cridarà al *Closest hit*. Si no s'ha trobat cap intersecció, es cridarà al programa *Miss*. En cas de estar monitoritzant excepcions i que es produeixi alguna, es cridarà el programa d'excepció. Al diagrama no apareix el programa *Bounding Box*, que es fa servir per calcular les estructures d'acceleració abans d'invocar el traçat de rajos.

5.2.2 Representació de l'escena

L'estructura d'escena que fa servir OptiX està formada pels nodes de jerarquia i pels objectes, veure figura 5.3.

Els **Nodes de jerarquia** són els que permeten representar l'escena com un graf, el qual controlarà el recorregut dels rajos. Per a suportar instanciació i compartir dades, els nodes poden tenir múltiples pares. N'hi ha de quatre tipus:

- *Grup*: serveix per a agregar diversos nodes de qualsevol tipus.
- *Grup de geometria*: són les fulles del graf d'escena. Han de tenir associats com a mínim una *Geometry Instance* i una estructura d'acceleració.
- *Transformació*: és un node amb una matriu de transformació afí 4×3 associada. Només pot tenir un fill, de tipus qualsevol. La transformació s'aplicarà als rajos que recorren el graf d'escena a través d'aquest node i a la geometria que hi continguin els nodes fills.
- *Selector*: té associats diversos fills i un programa de tipus *Selector visit* per decidir per quin d'ells ha de seguir el recorregut del raig actual.

Els **Objectes** són el volum més gran de les dades emmagatzemades a les fulles del graf. Contenen la geometria dels objectes i les operacions d'il·luminació i els materials. Com al cas anterior, poden tenir diversos pares per permetre compartir informació de geometria i materials. N'hi ha de tres tipus:

- *Geometria*: conté la llista de primitives geomètriques definint la posició i altres atributs com ara colors, normals i coordenades de textura. És on definirem els programes de *Intersection* i *Bounding Box*. Cal ressaltar que no és obligatori associar dades de geometria, doncs si volem fer-la procedural ja s'encarregarà el programa d'intersecció associat a aquest objecte de definir-la.

- *Material*: conté la informació necessària per a la il·luminació. És on definirem els programes de *Any hit* i *Closest hit*. A cada tipus de raig que tinguem definit a la nostra aplicació, podem associar-li un programa de cadascun dels dos tipus. Si no associem cap programa per a un parell de tipus d'intersecció i tipus de raig, no es farà res per a les interseccions detectades.

- *Geometry Instance*: una instància és una agrupació entre un objecte de geometria i un de material. D'aquesta manera, podem tenir totes les combinacions entre geometries i materials sense haver de replicar informació.

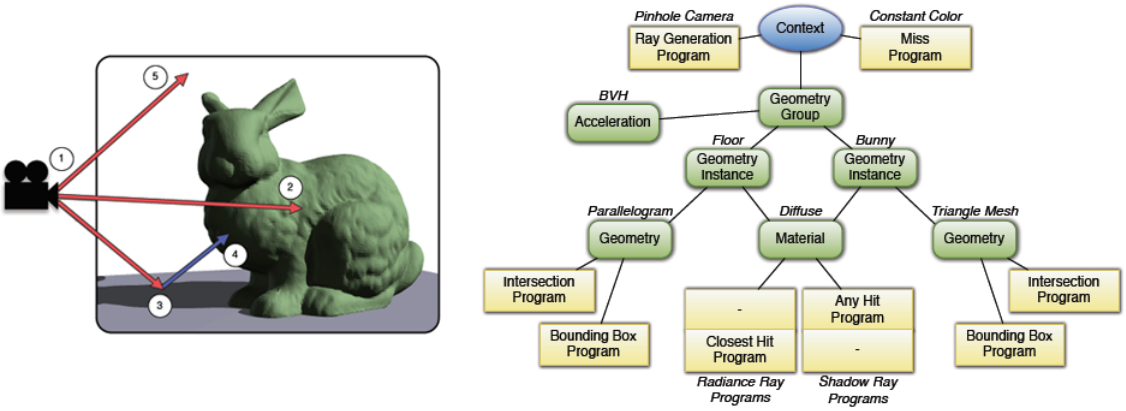

Figura 5.3: Exemple de graf d'escena, extret de [31]. 1: el programa de generació de rajos implementa la càmera. 2: el conill és una instància formada per la malla de triangles i el material. 3: el terra és una altra instància, que comparteix material amb el conill i té com a geometria un quadrilàter. 4: s'efectuen tests d'oclusió de la llum amb *shadow rays*, associats al programa *Any hit*. 5: si no hi ha intersecció, el programa de *Miss* defineix el blanc com a color de fons.

5.2.3 Estructures d'acceleració disponibles

Les estructures d'acceleració, a la versió més recent d'OptiX al moment d'escriure aquesta memòria (OptiX 2.1 RC1), no són programables. No obstant, podem triar entre diverses opcions cada vegada que volem associar-ne una a un node del graf d'escena. Les diferents estructures que tenim són:

- *BVH*: construeix una jerarquia de volums englobants. Se centra en qualitat més que en temps de construcció, i ofereix un bon balanç entre els *Split-BVH* i els *Median BVH*. Acostuma a ser la millor opció per als nodes grup.

- *Split-BVH*: és un BVH d'alta qualitat a canvi d'un major temps de construcció i més memòria. És la millor opció per a geometria estàtica.[38]

- *Median BVH*: produeix un BVH de qualitat mitjana fent servir un esquema de construcció més ràpid. És l'opció habitual per a geometria dinàmica.
- *Linear BVH*: és un esquema de construcció de BVH, sovint molt més ràpid que els *Median BVH*, però el rendiment del traçat de rajos acostuma a ser pitjor. [25]
- *KD-Tree*: estructura jeràrquica d'alta qualitat, en molts casos comparable al *Split-BVH*. És especialitzat per a geometria en malles de triangles.
- *No Acceleration*: constructor buit que no crearà cap estructura d'acceleració.

5.2.4 Model d'herència de paràmetres

Finalment, com s'ha comentat, OptiX busca minimitzar l'impacte de les dades basant-se en un **model d'herència** dels paràmetres. Aquest model ens defineix quina és la prioritat a l'hora d'heretar un paràmetre segons el tipus de programa en què ens trobem. La taula següent mostra aquest ordre:

Ray Generation	Programa	Context		
Exception	Programa	Context		
Closest Hit	Programa	Instància	Material	Context
Any Hit	Programa	Instància	Material	Context
Miss	Programa	Context		
Intersection	Programa	Instància	Geometria	Context
Bounding Box	Programa	Instància	Geometria	Context
Selector Visit	Programa	Node		

Taula 5.1: Prioritat d'herència segons el tipus de programa, de major a menor.

Un exemple d'aplicació d'aquest model seria fer servir des d'un programa una variable per a la qual volem donar un valor en funció de la instància i tenir un per defecte, en cas que la instància no el redefineixi.

5.3 OpenRL

OpenRL és l'acrònim de Open Ray Tracing Library, i es tracta d'una API creada per l'empresa Caustic Graphics[3] que pretén ser un estàndard per a incorporar Ray Tracing a les aplicacions gràfiques [8]. Les principals característiques són:

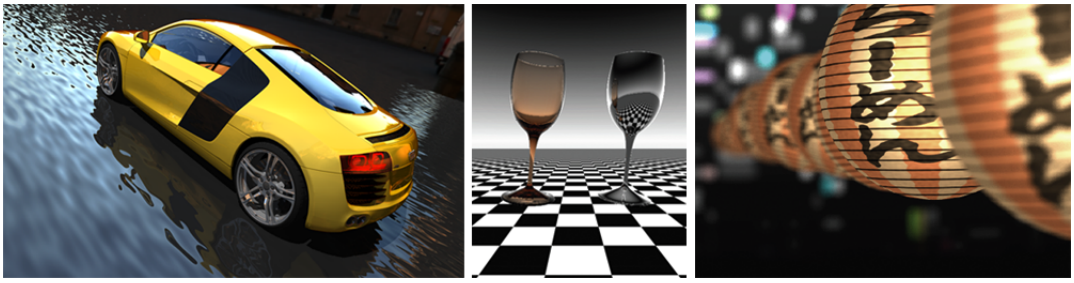

Figura 5.4: Exemples de render amb OpenRL, extrets de la documentació [8].

[3] http://www.caustic.com

- Portabilitat i eficiència entre diferents plataformes i hardware.
- No és un sistema de render complet, només proporciona el suport necessari per a llançar rajos i executar el codi dels shaders definits.
- No implementa cap algorisme de traçat de rajos en concret, proporciona els blocs bàsics i diverses funcionalitats per a que el programador implementi la tècnica que vulgui de manera fàcil i eficient.
- OpenRL es fa càrrec de: emmagatzemar i gestionar l'estructura de l'escena, emmagatzemar textures i buffers amb les dades i geometria, trobar les interseccions del raig amb l'escena i executar els shaders.

5.3.1 Fases d'execució

El programador haurà de crear i inicialitzar un context d'OpenRL. A continuació, se segueixen les següents fases:

1. **Data upload**: a diferència d'OpenGL, totes les dades de l'escena han de ser accessibles abans de poder començar a renderitzar. Hi ha quatre categories de dades que es poden fer servir:
   - *Geometria*: defineix la malla tridimensional i els atributs que s'associen a cadascun dels vèrtexs. Els rajos intersecaran amb les dades geomètriques de posició. S'ha d'enviar a OpenRL en forma de triangles fent servir *Vertex Buffer Objects*.
   - *Blocs uniformes*: dades de només lectura que seran accessibles des dels shaders i es mantindran constants durant tot el renderitzat del frame actual. El format de les dades és definit pel shader que hi accedirà.
   - *Textures*: són blocs de dades també de només lectura, però als que s'hi accedeix fent ús d'un *sampler*, el qual pot interpolar els valors de múltiples texels per filtrar i fer servir MIP Mapping. Les dades han d'estar en algun dels formats de textura suportats.
   - *Shaders*: programes descrits en codi RLSL que són compilats i enllaçats en un *program object*. N'hi ha de tres tipus: *vertex shader*, *frame shader* i *ray shader*.
2. **Scene setup**: les dades que hem enviat a l'etapa anterior s'han d'associar entre elles per a definir els diferents objectes de l'escena fent servir *Primitive objects*. Cadascuna d'aquestes primitives es pot pensar com l'encapsulat de tots els elements necessaris per a pintar un objecte concret: associació amb les dades de geometria, una associació amb un programa de shader, associacions amb dades uniformes i valors de les variables uniformes. A part, caldrà definir el *Framebuffer* i associar-li, com a mínim, una textura que contindrà el resultat final del render.
3. **Frame rendering**: un cop les dades s'han enviat i l'escena s'ha configurat, el *host* ha d'iniciar explícitament aquesta fase. A partir d'aquest moment, ja no es realitzarà més intervenció del *host* fins a que acabi el renderitzat. Primer s'executarà el *vertex shader* per cada vèrtex de cada primitiva de l'escena, i després s'executarà el *frame shader* associat al Framebuffer. Es considera que el renderitzat s'ha completat quan el *frame shader* s'ha executat un cop per cada píxel i totes les consultes d'intersecció de raig amb l'escena s'han resolt executant el programa associat a cadascuna.

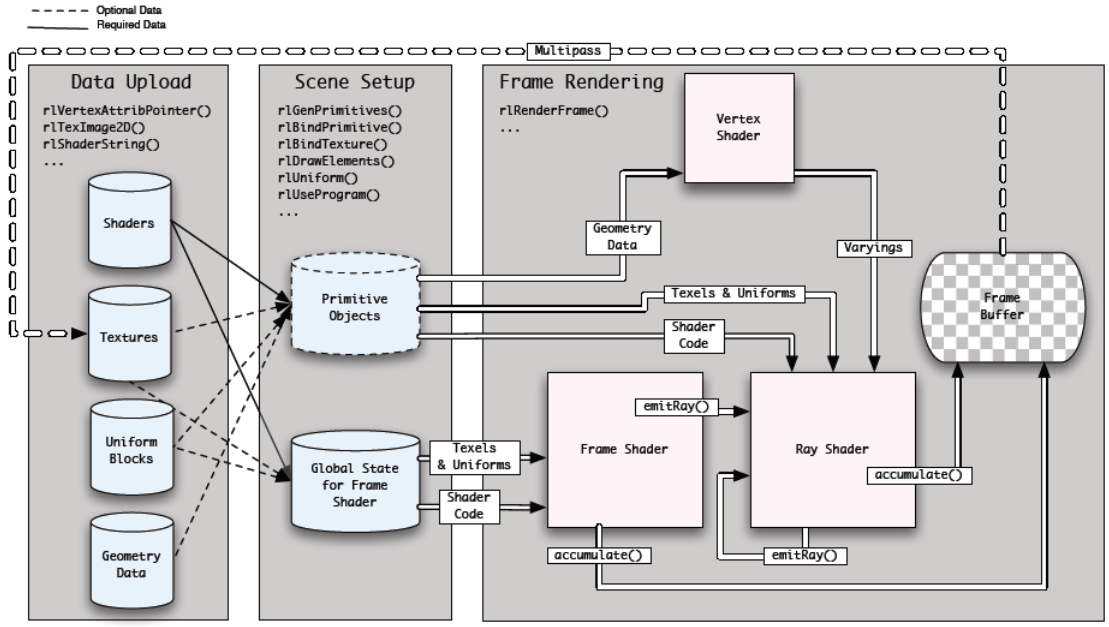

Figura 5.5: Diagrama de l'arquitectura d'OpenRL, extret de la documentació [8].

5.3.2 Tipus de shaders

Com hem vist al parlar de la primera fase d'execució, hi ha tres tipus diferents de *shaders*. Per a programar-los, es fa servir un derivat del OpenGL Shading Language (GLSL) [35] que han anomenat OpenRL Shading Language (RLSL). La sintaxi és gairebé idèntica a la de GLSL, té totes les característiques i funcions d'aquest i afegeix noves funcionalitats específiques per al traçat de rajos.

- **Vertex Shader**: per a cada objecte de l'escena, el seu vèrtex shader associat s'executa un cop per cadascun dels vèrtexs de la geometria que té associada. S'encarrega de transformar la posició continguda al *Vertex Buffer Object* a la posició final dins de l'escena i de definir els atributs que variaran sobre la superfície de l'objecte. Després d'aquest shader, l'atribut de posició i tots aquells altres atributs marcats com a `transformed` seran multiplicats per la matriu 4×4 de transformació associada a la primitiva per tal de determinar els seus valors finals.

- **Frame Shader**: a continuació, s'executa aquest *shader* un cop per cada píxel. És l'encarregat de construir els rajos primaris i emetre'ls, així com d'acumular valors al *Framebuffer* per implementar la càmera i efectes en espai d'imatge com, per exemple, filtres.

- **Ray Shader**: finalment, per cadascuna de les interseccions raig-geometria que es troben, s'executarà el *ray shader* associat a la primitiva. Aquest *shader* s'encarregarà d'implementar els efectes de la il·luminació i el material de la superfície al punt d'intersecció. La sortida pot ser un valor acumulat al *Framebuffer* i/o rajos addicionals, per exemple per a implementar reflexions o tests d'oclusió de la llum. Els rajos a OpenRL tenen el conjunt d'atributs definits següent:

  - Punt d'origen
  - Vector de direcció
  - Distància de l'origen a la intersecció (per a rajos d'entrada)
  - Distància màxima a recórrer (per a rajos de sortida)

- Nombre acumulat de rebots
- Si és un raig que es fa servir per a test d'oclusió (*flag* per a augmentar l'eficiència del test d'intersecció)
- Primitiva per defecte, amb el *shader* associat en cas de no intersecar l'escena.
- Tipus de raig, per a definir després dins del *shader* diferents comportaments.
- Diferencials de raig, per a saber l'expansió del píxel al punt d'intersecció.

Addicionalment, es poden afegir nous atributs al raig, com per exemple el color o la contribució que pot fer aquest raig.

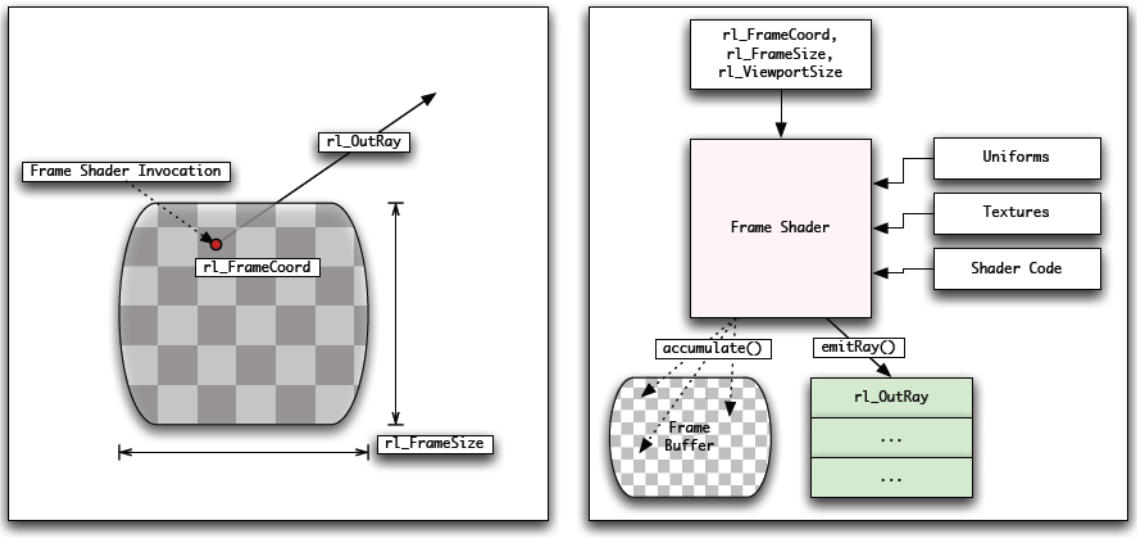

Figura 5.6: Entorn d'execució del Frame Shader. A l'esquerra, es mostra com es fa una invocació per cada píxel del *Framebuffer*. A la dreta, dades d'entrada i accions que pot realitzar. Extret de la documentació d'OpenRL [8].

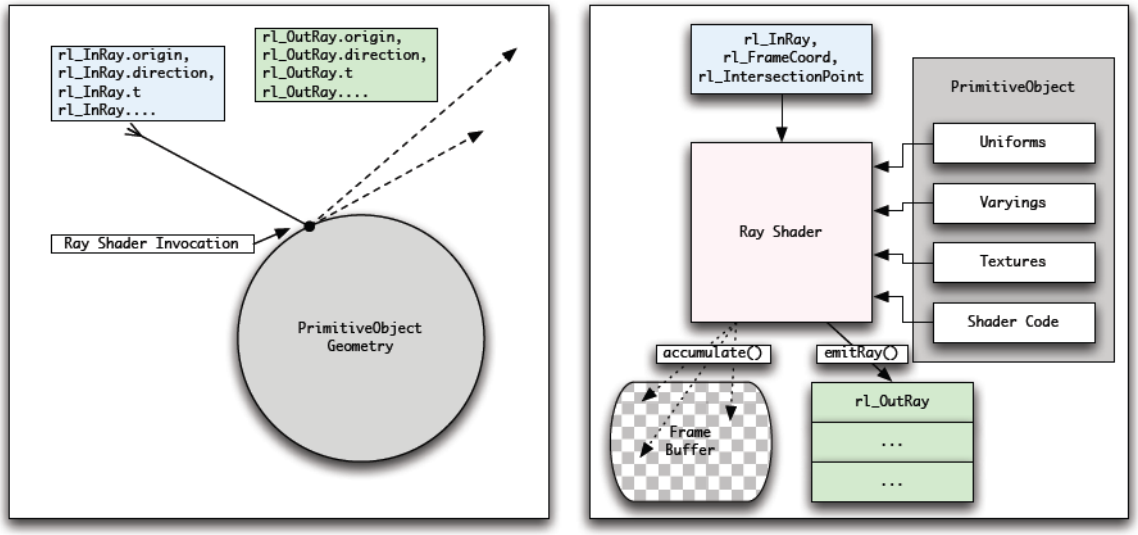

Figura 5.7: Entorn d'execució del Ray Shader. A l'esquerra, es mostra com es fa una invocació per cada intersecció amb una primitiva. A la dreta, dades d'entrada i accions que pot realitzar. Extret de la documentació d'OpenRL [8].

5.4 Altres Raytracers no interactius

Per acabar el capítol, farem breu menció d'altres Raytracers que són molt utilitzats però no ofereixen rendiments interactius, sinó que estan pensats per a realitzar el render d'un frame durant diversos minuts o, fins i tot, hores.

Un exemple és **Mental Ray**[4], un software desenvolupat per la companyia Mental Images, comprada per NVIDIA l'any 2007. S'ha fet servir com a render realista en diverses produccions de pel·lícules, entre elles *Tron: Legacy*, *Star Wars II*, *The Matrix Reloaded i Revolutions* o *Hulk*. Molts programes de modelat el porten incorporat per a fer els renders, com per exemple Autodesk Maya, Autodesk Softimage, 3D Studio Max, AutoCAD o Houdini. És el principal competidor de **RenderMan**[5], el sistema de render de la companyia de pel·lícules d'animació Pixar.

Figura 5.8: Exemples de renders amb Mental Ray.

Un altre exemple és **PBRT**[6], acrònim de Physically Based Ray Tracer, i que acompanya al llibre *Physically Based Rendering* [32]. L'objectiu del llibre és construir un Ray Tracer explicant progressivament tots els conceptes involucrats, començant pels tests d'intersecció i tractant temes com la física de la llum i els materials, mostreig fent servir mètodes de Monte Carlo i la implementació de diversos algorismes d'il·luminació global. El codi és lliure i s'acompanya de diversos algorismes implementats, amb la possibilitat de definir *plugins* per a ampliar-lo. **LuxRender**[7] és un programa basat en PBRT enfocat a artistes per a produir renders realistes.

Finalment, un dels Raytracers més antics que encara es fa servir és **POV-Ray**[8], acrònim de Persistence of Vision Raytracer. La primera versió que va fer el seu autor, David Kirk Buck, és dels anys 80. Amb el temps, ha anat evolucionant i incorporant els nous algorismes més rellevants en aquest camp. L'última versió és del 2009, i entre les seves característiques hi podem trobar: un llenguatge Turing-complet per a descriure l'escena, diversos tipus de fonts de llum, efectes atmosfèrics, ús de Photon Mapping per a les reflexions i refraccions, algorismes de radiositat o geometria constructiva de sòlids.

[4] http://www.mentalimages.com/products/mental-ray.html
[5] http://renderman.pixar.com/
[6] http://www.pbrt.org
[7] http://www.luxrender.net
[8] http://www.povray.org

Figura 5.9: Exemples de renders amb PBRT.

Figura 5.10: Exemples de renders amb POV-Ray.

Part II

Desenvolupament tècnic

Capítol 6

Algorisme d'il·luminació global per a entorns urbans

En aquest projecte, hem desenvolupat un nou algorisme per a la il·luminació global d'entorns urbans, basant-nos en el Photon Mapping. L'objectiu d'aquest capítol és oferir una breu descripció de les seves etapes, per tal d'entendre millor i tenir una visió general de l'algorisme quan s'expliqui cadascuna en detall als capítols següents.

La idea bàsica darrere el nostre algorisme és millorar l'estructura de dades que fa servir el Photon Mapping i adaptar-la i optimitzar-la tot el possible per a models de ciutats. Com veurem, aquests models tenen unes característiques concretes que fan que es considerin sovint models en "2.5D" en comptes de models 3D, ja que els podríem representar amb un mapa bidimensional en el qual, per a cada punt, tenim la seva alçada sobre el nivell del terra. Per tant, no té cap sentit utilitzar una estructura de dades com el *kd-tree*, que està pensada per a conjunts de punts qualssevol de l'espai tridimensional. La nostra estructura de dades es basarà en una parametrització en dues dimensions dels objectes de l'escena (les illes de cases o edificis) sobre unes textures, les quals després s'aplicaran als punts de l'escena que vulguem il·luminar.

Aprofitant que el Photon Mapping progressiu aconsegueix bons resultats interactius, ens basarem en aquesta implementació. Per tant, l'algorisme constarà també de tres etapes:

- **Pas de càmera** (capítol 8): es llançaran rajos des d'una càmera perspectiva situada a l'observador i es determinaran les superfícies visibles des d'aquest. Per a cada raig que s'hagi traçat, ens guardarem la informació necessària que ens permeti posteriorment fer els càlculs d'il·luminació del punt visible. Aquest pas només el farem quan es modifiqui algun paràmetre de la càmera o quan algun canvi a la configuració de l'algorisme ho requereixi.

- **Pas de fotons** (capítol 9): des de la font de llum es llançaran un nombre fixat de fotons que es propagaran per l'escena, fent que es reflecteixin a les superfícies especulars i que prenguin una nova direcció aleatòria a les difoses. Aquestes interaccions dels fotons amb l'escena les guardarem al mapa de fotons, que serà una textura amb les parametritzacions dels objectes. A cada iteració o *frame* es repetirà aquest procés.

- **Pas d'il·luminació** (capítol 10): finalment, es faran els càlculs per a il·luminar els punts visibles obtinguts al pas de càmera, fent servir la informació del mapa de fotons que s'ha generat al pas anterior. L'element clau d'aquest pas és aprofitar les parametritzacions que s'han fet dels objectes per tal d'evitar fer una cerca al *kd-tree* i convertir així el cost de cerca dels fotons més propers en accessos d'ordre constant.

Per a poder aplicar l'algorisme sobre qualsevol model de ciutat, necessitarem adaptar-lo prèviament a les nostres necessitats i generar les textures i informació addicional que després farem servir a l'algorisme. Els detalls de tot aquest **preprocés** es troben al capítol 7.

Hem escollit implementar l'algorisme fent servir **OptiX** (secció 5.2). Comparant-lo amb la seva millor alternativa actual, **OpenRL** (secció 5.3), hem vist que OptiX és més genèric i flexible: ens permet modificar més elements del funcionament del traçat de rajos, tot i que manté tancades les estructures d'acceleració del recorregut dels rajos. A més, fins i tot en aquest cas, se li pot dir quines estructures es vol que faci servir per a cada instància. Com ja hem comentat al capítol anterior, els rendiments que aconsegueix són bons comparant-lo amb els millors Raytracers actuals fets en CUDA i optimitzats a mà. Per tant, ens permet aprofitar la potència de CUDA i els càlculs a la GPU fent una bona abstracció dels detalls de baix nivell.

Capítol 7

Preprocés

El preprocés inclou tot aquell tractament que farem al model i a la resta de dades associades a aquest, com per exemple les textures, per tal d'adequar-lo a les necessitats específiques del nostre algorisme, superar les limitacions imposades pel hardware i per OptiX, o senzillament per millorar el rendiment. Veurem el cas concret del model de Barcelona, fet per Tele Atlas, que s'ha fet servir durant la realització del projecte, però amb pocs canvis es podria adaptar a altres models de ciutats.

7.1 Model de la ciutat

Aquest apartat descriurà tots aquells punts del preprocés que facin referència al tractament del model de la ciutat. Bàsicament, voldrem carregar-lo, ser capaços de crear models derivats amb només algunes zones concretes i, tot i que no és un requisit per al posterior funcionament de l'algorisme, ens interessarà simplificar-lo.

7.1.1 Carregador del model original

Com és lògic, el primer que ens caldrà per a començar a treballar amb el model és ser capaços de llegir-lo i mostrar-lo per pantalla. El model del qual disposàvem no ens venia donat en cap format conegut com per exemple OBJ o PLY, sinó que consistia en un fitxer de text pla amb el format següent:

nom del model	string
nombre d'objectes	enter positiu
per cada objecte	
nombre de cares	enter positiu
per cada cara	
tipus de primitiva	enter positiu (GLenum)
nom de la textura	string
nombre de vèrtexs	enter positiu
per cada vèrtex	
posicio (x, y, z)	3 floats
coordenades de textura (s, t)	2 floats

Per tant, fer un carregador és molt senzill: obrirem el fitxer, llegirem quants objectes hi ha i per cada objecte llegirem les seves cares i les anirem omplint amb la informació. Cada cara ve descrita per un dels tipus de primitiva d'OpenGL, que s'identifiquen amb un enter, el nom de la textura associada i el nombre de vèrtexs que conté. Per a cada vèrtex

tenim 5 nombres en coma flotant que corresponen a les tres coordenades de posició i a les dues de textura.

Cada objecte individual del model pot correspondre o bé a un bloc de pisos o casa individual, o bé a una illa de cases sencera sense tenir els edificis de manera separada. Segurament, es podria mirar de quina manera estan donades les cares d'una illa de cases i es podria separar fàcilment en els seus blocs de pisos individuals. No obstant, no hem necessitat fer-ho en cap moment i per això s'han tractat sempre com indivisibles. Per tant, d'ara en endavant farem referència als objectes anomenant-los edificis, indistintament de si són blocs individuals o una illa formada per diversos blocs "inseparables". Quan calgui parlar d'un dels dos tipus en concret i evitar confusions, es farà explícit de quin es tracta.

Figura 7.1: Visualització de tot el model de Barcelona amb només la geometria, sense textures.

Una altra funcionalitat que s'ha afegit al carregador és poder seleccionar un conjunt d'edificis, ja sigui per identificador o pels que es troben dins d'una regió quadrada del model, i desar aquesta selecció a un model a part. Així es poden obtenir models més petits que fan més còmode la visualització de zones concretes i agilitzen el procés de prova del programa.

7.1.2 Triangulació de les cares

Si analitzem les cares dels objectes, veiem que les hi ha de quatre tipus diferents: triangles, tires de triangles, ventalls de triangles i polígons (convexos). Per a simplificar el programa de col·lisions que ens caldrà implementar a OptiX, i evitar en la mesura del possible els condicionals, hem convertit totes les primitives a triangles.

La representació que tenim per a un objecte és un vector de vèrtexs i un vector de cares. Cadascun dels vèrtexs té els atributs de posició, normal, coordenades de textura i color. Cada cara té el nom de la textura i una llista d'enters positius que corresponen a posicions (índexs) del vector de vèrtexs de tot l'objecte.

Amb aquesta representació, triangular el model és trivial ja que només ens cal modificar el vector d'índexs que té la cara segons el tipus de primitiva:

- Pels ventalls de triangles, tenim una seqüència d'índexs v_1, v_2, v_3, v_4, ..., v_n. Els

triangles que es formen són: (v_1, v_2, v_3), (v_1, v_3, v_4), ..., (v_1, v_{n-1}, v_n). Per tant, només ens caldrà modificar el vector d'índexs per a que contingui explícitament tots aquests triangles un a un. Si teníem n índexs, el vector resultant tindrà $3(n-2)$ índexs.

- Pels polígons, donat que sempre són convexos, podem seleccionar un índex qualsevol (per exemple el primer) i procedir de la mateixa manera que s'ha fet amb els ventalls de triangles. També passarem de n a $3(n-2)$ índexs.

- El cas de les tires de triangles requereix alguna consideració addicional. Si tenim de nou la seqüència d'índexs v_1, v_2, v_3, v_4, ..., v_n, ara els triangles que es formen són (v_1, v_2, v_3), (v_3, v_2, v_4), (v_3, v_4, v_5), ... Fixem-nos que cal anar invertint cada dos triangles l'ordre dels dos vèrtexs anteriors per tal de mantenir totes les normals orientades cap al mateix costat.

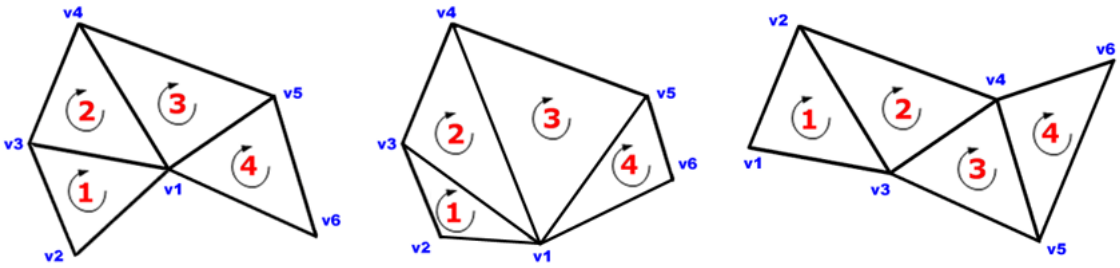

Figura 7.2: Esquema que mostra com es triangulen les diferents primitives de les que consta el model: el ventall de triangles, els polígons convexos i les tires de triangles.

7.1.3 Edificis no texturats

Quan vam visualitzar el model amb textura, ens vàrem adonar que hi havia un conjunt d'edificis pels quals no s'havia assignat un nom de textura a les parets laterals. Tots aquests edificis formen part de les zones més allunyades del centre, de manera que eliminar-los és una opció que no deixarà buits sense edificis enmig d'altres edificis. Una altra opció seria assignar textures de manera aleatòria o seguint algun criteri procedural. La figura 7.3 mostra la zona de Barcelona texturada.

Figura 7.3: Imatge dels edificis texturats (en verd) i els no texturats (en vermell).

7.1.4 Eliminació de polígons no visibles

L'última modificació que s'ha fet sobre el model és l'eliminació de polígons no visibles. El model contenia un nombre elevat de polígons que, amb la visualització que es farà, mai seran visibles i, per tant, es poden eliminar i obtenir un model més reduït. Un exemple de situació on clarament hi ha polígons no visibles és a les illes de cases. Les parets laterals dels blocs de pisos, és a dir, les que separen dos blocs un al costat de l'altre, són cares que arriben del sostre al terra, sovint amb diversos triangles. A partir del punt en què els dos edificis tenen la mateixa alçada i fins al terra, tots els triangles d'aquestes parets mai seran visibles ja que no hi ha façanes transparents. Per tant, els podem eliminar sense apreciar cap canvi al model quan el visualitzem.

L'algorisme que s'ha fet servir per a eliminar els polígons està basat en la visibilitat d'aquests. Per cada objecte, fem diversos passos de pintat amb una càmera que miri al centre de l'objecte cada vegada des d'un punt diferent sobre la superfície de l'esfera englobant. Cada polígon es pinta fent servir *fals color*, és a dir, codificant el seu identificador com un color en RGB. Després de cada pas de pintat, s'analitza la imatge obtinguda i per tots aquells píxels que no siguin negres (color de fons) es calcula l'identificador associat i s'afegeix a un conjunt d'identificadors trobats. Quan s'han fet tots els passos de pintat, tots els polígons amb un identificador que no formi part del conjunt es podran eliminar ja que mai s'han vist. Tot i que aquest algorisme és només una aproximació del càlcul de la visibilitat, és més que suficient pels nostres propòsits.

Cal aclarir que no pintem només l'objecte, també afegim un pla sota d'ell que representarà el terra. Sinó, com els edificis d'aquest model no són tancats per la part inferior, des de les càmeres situades a sota l'objecte veuríem tots els polígons de l'interior que estem intentant eliminar.

La taula següent mostra els resultats obtinguts pel model de Barcelona (només agafant com a model d'entrada els edificis texturats, veure figura 7.3).

	Model original	**Model reduït**	**Reducció**
vèrtexs	4 966 414	3 606 142	72.6%
cares	1 217 132	876 497	72.1%
triangles	2 525 190	1 836 204	72.7%

Taula 7.1: Resultats de l'eliminació de polígons no visibles del model de Barcelona.

7.2 Textures

El següent conjunt de dades a processar, un cop tenim ja el model preparat, són les textures. Juntament amb el model, es disposava d'un total de 23 839 textures amb diferents models de parets, finestres, portes, comerços... i cadascun en diversos colors. Si analitzem quines es fan servir realment al model de Barcelona veiem que només 5 289 d'entre totes les anteriors són útils.

7.2.1 Atles de textures

Per a fer un visualitzador amb OpenGL tindríem diverses opcions per a tractar amb aquesta quantitat de textures, com podria ser fer servir un gestor de textures amb una memòria

màxima permesa d'ocupar, i que les anés carregant i descarregant a mesura que es necessitin per a visualitzar els edificis que es volen mostrar. A més a més, totes les que estiguessin actives al moment es podrien posar en un *texture array*[1] i accedir-hi a totes juntes fent servir un únic *sampler*. Malauradament, la versió 2.1 d'OptiX no té el suport complet per a *texture arrays*. Fins i tot, si el tingués, degut a la naturalesa dels Raytracers en comparació als paradigmes projectius com el d'OpenGL, OptiX necessitaria disposar de totes les dades, tant de geometria com de textures, abans de començar un pas de traçat de rajos. Per tant, és necessari que puguem posar-ho tot a la memòria de la targeta gràfica.

Després d'analitzar detingudament les textures que estava fent servir el model de Barcelona, es va optar per fer servir un *atles de textura*. Un atles de textura és una textura 2D que conté d'altres textures. D'aquesta manera, si recalculem apropiadament les coordenades de textura dels vèrtexs i els hi canviem apropiadament la textura que fan servir, podem reduir molt el nombre de textures necessàries. Si, a més a més, reduïm la mida de les textures originals quan les copiem a l'atles de textura, estarem també reduint la memòria ocupada a la targeta gràfica. El principal problema de fer servir atles de textura és que, degut a la interpolació bilineal, quan agafem texels que toquen a una textura veïna de l'atles apareixen *seams* o costures visibles sobre el model.

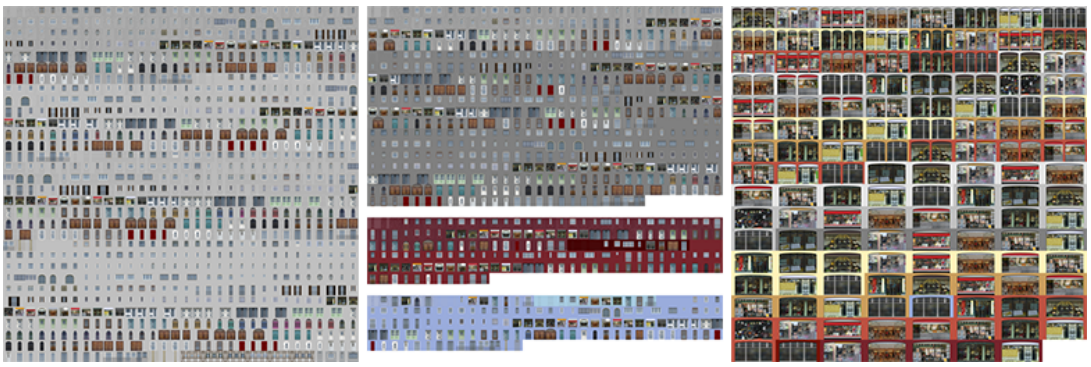

Figura 7.4: Tres exemples dels atles de textura que es faran servir per a visualitzar Barcelona. Els dos primers contenen textures de 32×32 píxels i el tercer, de comerços a 128×64.

Les textures originals eren majoritàriament de 512×512 píxels cadascuna, amb variants més primes de 256×512 o 128×512. En alguns pocs casos teníem textures allargades de 1024×512. Degut a les limitacions imposades per la memòria de la targeta gràfica, s'ha decidit construir atles de textures a partir de les textures reduïdes a 32×32 (i variants proporcionals). Les textures de comerços, originalment de 1024×512, s'han reduït a 128×64 píxels per preservar millor els detalls de l'aparador. Hi ha també textures de comerços i altres elements com per exemple portes de garatges, amb mida original de 512×512, que s'han passat a 64×64.

En total, s'han creat 9 atles de textures, cadascun d'ells de mida 1024×1024 píxels. Els 6 primers contenen parets, portes, finestres, etc. Cadascun d'ells podria contenir fins a 1024 textures reduïdes. Tot i això, per a minimitzar l'efecte de les costures s'han agrupat les textures per color i s'han separat quan calia posar més d'un color per atles. Identificar el color ha estat senzill, ja que el nom de cada textura incloïa un codi numèric descrivint el tipus de textura, color i model concret. Tenim també 2 atles de textura amb comerços de

[1] Un *texture array* es pot veure literalment com un vector de textures 2D. És en realitat una textura 3D per a la qual accedim fent servir com a 3a coordenada l'índex de la textura desitjada i no fa interpolació de cap tipus entre textures consecutives.

128×64, podent-ne posar 128 per textura. Finalment, un últim atles agrupa les de mida 64×64 corresponents a comerços i altres elements.

7.2.2 Codificació de superfícies especulars

Ara per ara, tenim un model amb només geometria i un conjunt de textures amb només color. En cap moment es té informació sobre com d'especular és una superfície. Ens agradaria poder mostrar algunes superfícies especulars amb reflexions, per exemple els vidres dels aparadors de les botigues o els edificis d'oficines on tota la façana són finestres. Per tal de poder indicar al visualitzador quines superfícies reflectiran la llum i quines seran difoses, s'ha fet servir el canal alfa de les textures. Quan es construeix l'atles de textures s'afegeix al canal alfa informació sobre la especularitat. Concretament, el posarem a 0 a tots aquells *texels* que vulguem marcar com especulars, i a d'altres valors si no ho serà.

A les textures donades no sembla haver cap mètode senzill que ens permeti decidir quines zones són vidres i que voldrem que reflecteixin la llum. Per sort, com ja s'ha mencionat abans, les textures tenen un codi numèric que indiquen el tipus del qual es tracta. Prenent uns pocs tipus significatius de textura de comerç, i gràcies a que aquests estan repetits en diferents colors, podem crear a mà una plantilla o màscara que indiqui quines zones volem marcar com especulars. A l'hora de construir els atles de textures, si al nom identifiquem que és un comerç, carreguem la plantilla corresponent al seu tipus i copiarem el canal alfa d'aquesta. Altrament, si no té cap plantilla posarem el canal alfa tot a 255, pel que cap regió serà especular.

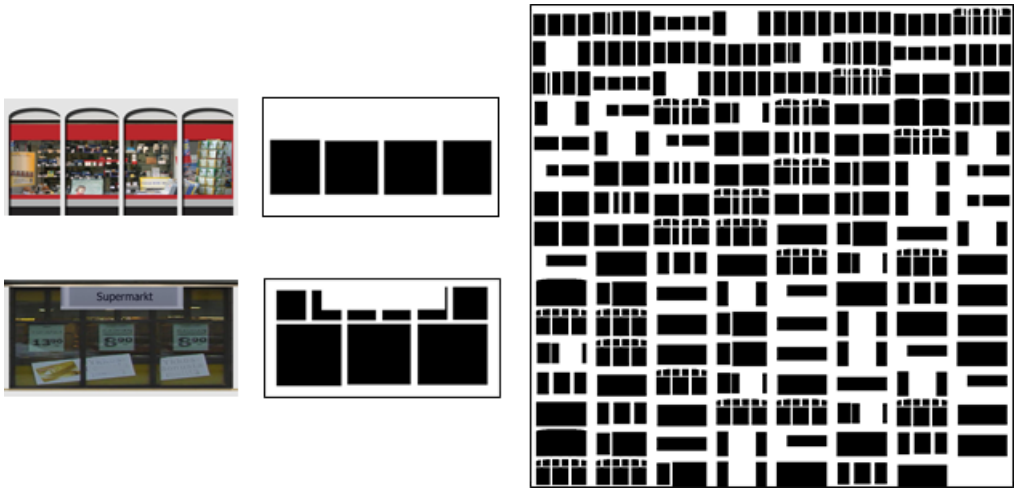

Figura 7.5: Composició del canal *alpha* als atles de textures. A l'esquerra es mostren dues textures originals, de mida 1024×512 amb la seva màscara de transparència que hem fet. A la dreta, es mostra el canal *alpha* de la 3a textura de la imatge 7.4.

7.3 Ortofotografies

Hem vist com tractar les textures per a les façanes. Ens falta veure com texturem el terra i els terrats dels edificis. Per a fer-ho, disposem de les diverses ortofotografies[2] obtingudes

[2]Una ortofotografia (abreviant, ortofoto) és una fotografia d'un terreny on s'ha aconseguit corregir la perspectiva i mostrar els elements en projecció ortogonal. Normalment s'obtenen aplicant un procés de correcció a un conjunt de fotografies aèries fetes des d'un avió o satèl·lit.

de l'Institut Cartogràfic de Catalunya (ICC). En total, són 18 fotografies a escala 1:5000 (50 cm per píxel) de la zona de Barcelona. Per a cada ortofoto es tenen les coordenades a les quals correspon.

7.3.1 Generació de la textura per a terres i terrats

Seguint la idea d'agrupar-les i tenir el mínim possible de textures, aquest cop farem una única textura final. A més a més, per a simplificar els accessos no serà un atles de textures sinó que ens aprofitarem que la ortofoto es pot considerar la projecció sobre el pla del terra. Per tant, la projecció del vèrtex sobre aquest pla (pel cas del model de Barcelona, consisteix en descartar la coordenada Y) ens podrà donar les coordenades de textura si sabem quin punt correspon a les coordenades $(0, 0)$ i quin a les $(1, 1)$.

Donat qualsevol subconjunt dels d'edificis del model sencer de Barcelona, és molt senzill generar la ortofoto. Primer de tot, calculem la capsa englobant d'aquest conjunt d'edificis i ens quedem amb la projecció sobre el pla del terra. Posem una càmera ortogonal mirant cap al centre d'aquesta capsa projectada de tal manera que les cantonades del *viewport* coincideixin amb les de la capsa englobant. Sabem que cada ortofoto és un paral·lelogram sobre el terra i sabem les coordenades dels seus quatre vèrtexs. Per tant, si detectem que interseca amb la projecció de la capsa englobant dels edificis, carreguem la textura i pintem el seu quadrat texturant-lo. Un cop analitzades totes les ortofotos i pintat les que siguin necessàries, tindrem la composició amb tota la projecció sobre el terra de la capsa englobant texturada.

La mida de la ortofoto composada que generem és sempre de 4096×4096. Per tant, la resolució sobre el model dependrà de com de gran sigui aquest en quan a extensió sobre el pla del terra. Per a models que ocupin poca zona, tindrem molt bona resolució al terra, mentre que per models de tota la ciutat serà bastant baixa.

Figura 7.6: Exemple d'ortofoto composada a partir de diverses fulles, senyalades sobre la imatge.

Un problema que tenen aquestes ortofotografies que hem fet servir és que per a alguns edificis es veu una ombra bastant fosca. Això pot despistar-nos quan intentem veure la nostra simulació d'il·luminació global, ja que podem confondre l'ombra generada i la de la ortofoto. Es podria intentar corregir l'ombra, però no creiem que es puguin obtenir bons resultats amb algun mètode senzill. Com les textures en sí no formen part de l'objectiu principal del projecte, s'ha optat per tenir una opció al programa que permeti activar i desactivar l'ús de la ortofoto per al terra. Pels terrats dels edificis s'ha deixat sempre activa.

Figura 7.7: Detall de l'ortofoto per una zona de l'Eixample on es poden veure molt bé les ombres que idealment no haurien d'existir.

7.3.2 Mapa d'alçades

Com veurem al capítol 10, necessitarem tenir un mapa d'alçades i un mapa amb els identificadors dels edificis codificats en fals color realitzats fent servir projecció ortogonal. Aquest dos mapes es poden combinar en un de sol si creem una textura RGBA on els tres canals de color ens permetin emmagatzemar l'identificador i el canal *alpha* l'alçada.

Per a generar el mapa d'identificadors no hi ha cap complicació. Senzillament posem una càmera ortogonal mirant al centre de la capsa englobant dels edificis des de dalt i pintem cadascun d'ells amb el color que codifica l'identificador, deixant el color negre per al fons. Per al mapa d'alçades, farem servir un *fragment shader* que pinti als canals RGB el color que ha rebut, ja que porta l'identificador, i al *alpha* l'alçada normalitzada entre 0 i 1, on 0 serà el terra i 1 l'alçada de l'edifici més alt. Fixem-nos que estem quantitzant el nombre possible d'alçades que serem capaços de distingir a 256, ja que es fan servir 8 bits per a cada canal. De la mateixa manera, "només" serem capaços de codificar $2^{24} - 1 = 16\,777\,215$ identificadors d'edifici. Com Barcelona sencera en té 13 588, això no ens suposa cap problema.

Per tant, donats un identificador `ID` i una alçada y, la codificació dins d'un píxel de la textura seria:

```
r = ID & 0x000000FF;
g = ID & 0x0000FF00;
b = ID & 0x00FF0000;
a = 255*(y - ymin)/(ymax - ymin);
```

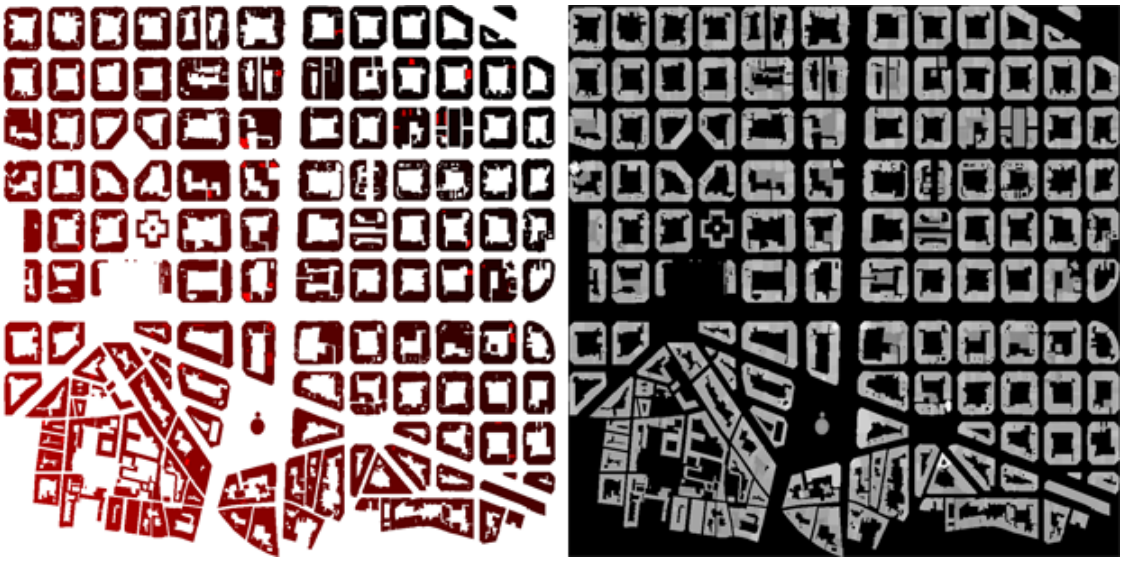

Figura 7.8: Imatge amb el mapa d'identificadors (esquerra) que correspondria a les components RGB i el mapa d'alçades (dreta) que correspondria a la component *Alpha* de la textura generada.

7.4 Façanes

Per últim, l'altre element que el preprocés haurà de generar són dues textures RGBA que continguin una representació a baixa resolució de les façanes de les diverses illes de cases, codificant el color als canals RGB i la distància entre la façana i el centre de l'edifici al canal *Alpha*. Com veurem al capítol 9, farem servir el color que llegim d'aquestes textures per a calcular l'atenuació i propagació de color a les interaccions difoses dels fotons. La distància al centre ens servirà per a evitar barrejar fotons contigus del nostre mapa/textura de fotons que en realitat corresponguin a parets diferenciades en profunditat. Això últim s'explicarà més en detall al capítol 10.

7.4.1 Parametrització

Per a implementar la nostra estructura del mapa de fotons i per a generar aquestes textures ens cal definir una parametrització de les façanes dels edificis, en aquest cas referint-nos a illes de cases senceres. Per senzillesa d'implementació i simplicitat del càlcul, farem servir una **parametrització cilíndrica**. La figura 7.9 mostra que és una aproximació de les illes de cases bastant raonable, ja que la majoria d'elles són bastant quadrades o rectangulars. A les illes de cases que tinguin pati interior podem fer dues parametritzacions, una per a les façanes que donen a l'exterior (carrer) i una altra per a les façanes que donen al pati interior.

Obtenir la parametrització d'una illa de cases és molt senzill. Imaginem que projectem sobre un cilindre les façanes, per exemple les exteriors, i retallem aquest cilindre per l'eix vertical, desplegant-lo en forma de rectangle. La base d'aquest rectangle serà el rang d'angles $[-\pi, \pi]$ i l'alçada del rectangle serà el rang $y_{min}\, y_{max}$. Normalment ens interessarà normalitzar aquests intervals i expressar-los entre 0 i 1.

A partir d'una illa de cases, per a obtenir el rectangle $[0, 1] \times [0, 1]$ corresponent a la seva parametrització cilíndrica ens farà falta saber les coordenades del centre c i els

valors y_{min} i y_{max} de la seva capsa englobant. A partir d'aquestes dades, les coordenades cilíndriques (s,t) d'un punt (x,y,z) les trobem aplicant:

$$\begin{aligned} s &= \frac{\arctan(x/z)+\pi}{2\pi} \\ t &= \frac{y-y_{min}}{y_{max}-y_{min}} \end{aligned}$$

Figura 7.9: Parametrització cilíndrica de les illes de cases superposada a la projecció sobre el pla del terra d'aquestes. El punt vermell mostra el centre.

7.4.2 Col·locació dels edificis a la textura

Acabem de veure com obtenir una parametrització cilíndrica normalitzada entre 0 i 1 de les façanes illes de cases. És evident que tenir tots dos eixos normalitzats al mateix rang ens està introduint una certa distorsió: un quadrat en espai paramètric correspondrà a un rectangle sobre les façanes. Fàcilment podem imaginar que aquest rectangle serà estirat en horitzontal, ja que el perímetre d'una illa de cases gairebé sempre serà molt major a l'alçada. Volem evitar que la manera com col·loquem els edificis sobre la textura introdueixi aquestes distorsions en la mesura que sigui possible.

Podem cercar i comprovar amb dades numèriques reals aquesta afirmació. A Barcelona la majoria d'illes de l'eixample tenen un perímetre al voltant dels 400 m, mentre que els tres edificis més alts són l'Hotel Arts i la Torre Mapfre, tots dos amb 154 m d'alçada, seguits de la Torre Agbar amb 145 m. Per tant, això corrobora el que hem comentat sobre l'allargament horitzontal dels quadrats en espai paramètric. Analitzant la relació entre perímetre i alçada de la capsa englobant de les illes de cases del nostre model arribem a la relació aproximada de perimetre $= 10 \times$ alçada.

Col·locarem els edificis en una matriu amb r files i c columnes. Si tenim N edificis en total, volem $N \leq r \times c$. Per la relació que acabem de calcular, $r = 10c$. Per tant:

$$N \leq r \times c \implies N \leq 10\,c^2 \implies c = \left\lceil \sqrt{N/10} \right\rceil$$

Observem que $r = 10c$ correspon a la relació calculada entre perímetre i alçada, que no té per què ser 10:1 sinó la que calgui a cada model. Per tant, una formula més genèrica seria $c = \left\lceil \sqrt{N/\rho} \right\rceil$, on ρ és la mitjana de la relació entre el perímetre i l'alçada.

7.4.3 Generació de les textures de façanes

Un cop sabem parametritzar cada illa de cases i sabem les dimensions sobre la textura que haurà de tenir cada parametrització, podem procedir a generar la textura que volíem. El principal problema que tenim inicialment és assegurar que les façanes que pintem sobre la textura parametritzada són, en realitat, les més exteriors que donen al carrer o les més interiors si volem les orientades al pati interior. Aquest problema és en realitat el mateix que tenim quan pintem diversos objectes sobre el pla de la càmera projectant-los seqüencialment i hem de saber quin es troba davant d'un altre. Per tant, aprofitarem que el hardware gràfic actual soluciona aquest problema amb el *z-test*.

Per a una primera versió de la generació de les textures farem servir el *vertex shader* i *fragment shader* (veure apèndix A per a una breu descripció sobre el *pipeline* d'OpenGL). El *vertex shader* rebrà com a variables uniformes el centre, l'alçada màxima i mínima i el radi de la capsa de l'edifici actual. Cada vèrtex que rebi amb coordenades (x, y, z) serà transformat a (s, t) fent servir les expressions de l'apartat 7.4.1. Cal tenir en compte que les coordenades de sortida que ha de generar un *vertex shader* són en espai normalitzat a $[-1, 1]$ i que no hem d'ocupar tota la textura sinó només la regió que correspongui a l'illa de cases que estem projectant. Tenint això en compte, escalarem i traslladarem (s, t) apropiadament per a que vagi a parar a la subzona de tota la textura de façanes apropiada. A més, la coordenada z de sortida la calcularem com la distància al centre de la capsa. Per tant, si estem pintant les façanes exteriors voldrem configurar el *z-test* per a que passin aquells fragments amb z major, mentre que si volem les interiors deixarem passar les z menors. Posteriorment, el *fragment shader* s'encarregarà únicament de pintar a la textura final amb el color que llegirà de les textures d'elements de les façanes que hem vist a la secció 7.2 i assignarà al canal *alpha* el valor de la profunditat.

La versió que acabem de descriure no funcionarà correctament tot i que en principi sembli correcta. Tindrem un error a causa del funcionament de l'algorisme de rasteritzat i la discontinuïtat que introduïm al "tallar" el cilindre verticalment i desplegar-lo. Si un triangle del model és creuat per la recta de tall del cilindre, el resultat que veurem serà incorrecte. La figura 7.10 il·lustra aquest cas.

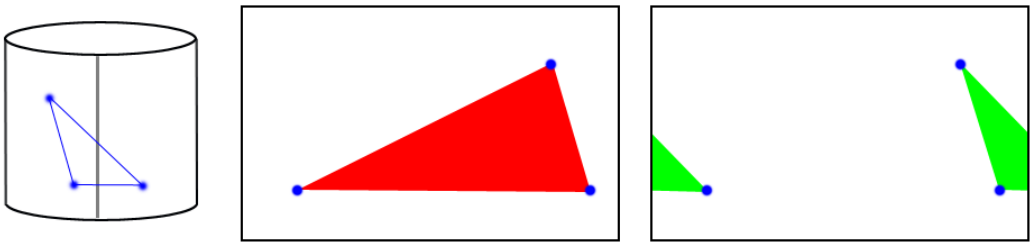

Figura 7.10: A l'esquerra, es mostra un triangle que creua el tall del cilindre. Al centre, el resultat erroni que obtenim al rasteritzar el triangle que formen els punts al canviar de coordenades. A la dreta, el resultat que ens agradaria obtenir.

La solució que hem trobat és fer servir el *geometry shader* per detectar aquest tipus de triangles que donaran errors, ja que des d'aquest *shader* tenim accés a la informació dels tres vèrtexs. Si el triangle no creua la recta de tall, procedim com abans. En cas contrari, crearem dos triangles de tal manera que un sobresurti per l'esquerra del rectangle on fem la parametrització i l'altre per la dreta. Fixem-nos que els nous vèrtexs s'obtindran sumant o restant l'amplada total que té assignada l'interval $[-\pi, \pi]$. El *fragment shader* haurà de ser capaç de detectar aquests fragments que surten de l'àrea assignada i descartar-los, o

estarem pintant a sobre les façanes de les illes contigües a la textura. La figura 7.11 mostra aquesta idea.

Figura 7.11: Esquema del funcionament de la solució amb el *geometry shader*. Es creen dos triangles, un a cada costat i després el *fragment shader* s'encarregarà de descartar els fragments sobrants, que al dibuix es mostren vermells.

La següent figura mostra els resultats erronis que obteníem sense tractar els triangles conflictius i l'aspecte correcte que mostren les façanes un cop solucionem aquest problema.

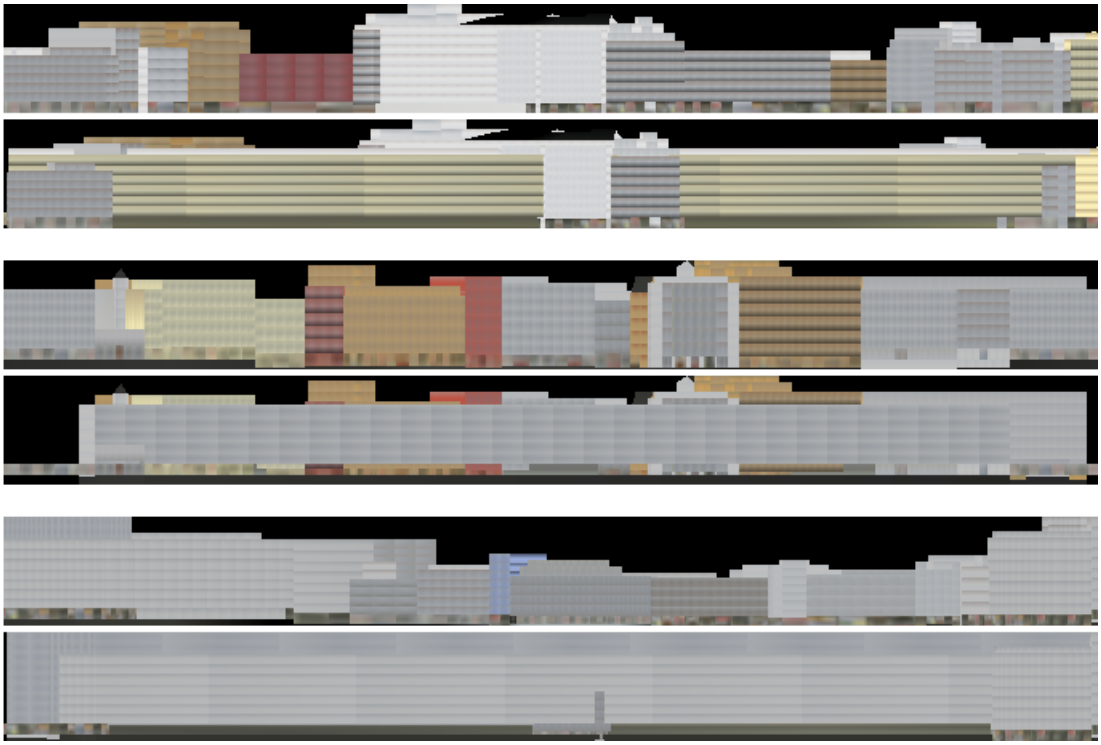

Figura 7.12: Tres parells d'exemples de parametritzacions, cadascun amb la correcta (primera imatge de cada parell) i incorrecta (segona imatge de cada parell).

La textura obtinguda té l'aspecte que mostren els següents exemples de la figura 7.13, fets agafant diferents zones del model sencer. La columna esquerra mostra els canals de color i la dreta el canal *alpha*. La primera imatge correspon a les façanes frontals d'un subconjunt de 44 edificis pel centre de Barcelona, i la segona a les seves façanes interiors. La tercera imatge és per a un conjunt de 1425 edificis, majoritàriament de l'Eixample.

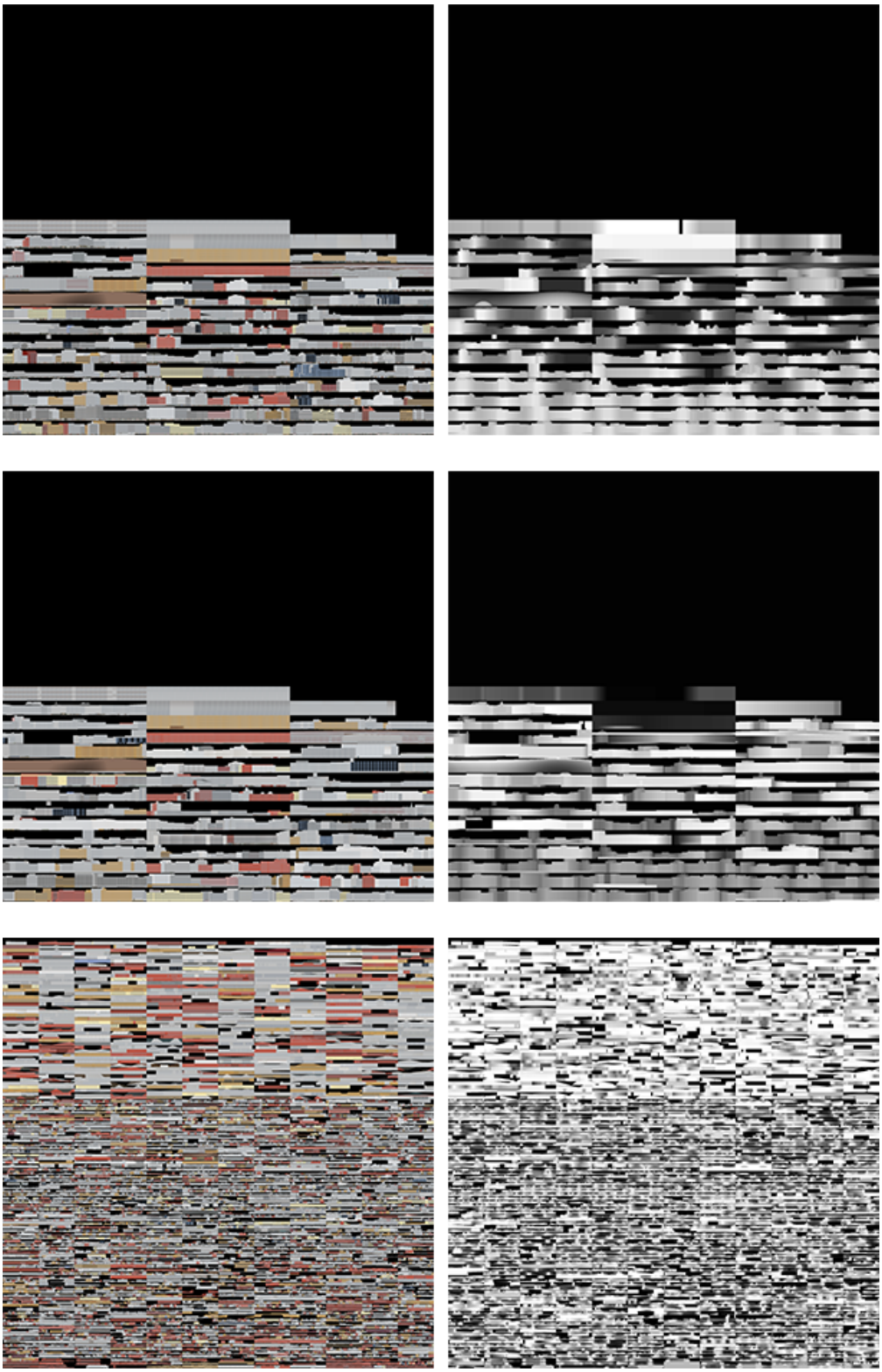

Figura 7.13: Exemples de textures amb la parametrització de les façanes.

Textura	**Mida**	**Pas utilització**	**Descripció**
Atles de textures (diversos)	1024×1024	Pas de càmera	Conté les textures amb els elements dels quals es composen les façanes (tipus de paret, finestres, portes, comerços...). Al canal *alpha* es codifica si la superfície serà especular.
Ortofoto	4096×4096	Pas de càmera	Ortofotografia composada per a la projecció sobre el pla horitzontal de la capsa englobant de l'escena.
Mapa d'alçades i identificadors	1024×1024	Pas d'il·luminació	Imatge obtinguda de la projecció ortogonal dels edificis sobre el pla horitzontal de l'escena. Als canals RGB es codifica l'identificador de l'edifici, el canal *alpha* conté l'alçada.
Textura de façanes	1024×1024	Pas de fotons i pas d'il·luminació	Imatge obtinguda de la parametrització cilíndrica dels edificis. Als canals RGB té les façanes en color, i al canal *alpha* la distància al centre de la capsa englobant de l'edifici.

Taula 7.2: Resum de les textures utilitzades a l'algorisme.

Capítol 8

Pas de càmera

El primer pas del nostre algorisme és l'anomenat **pas de càmera**, que consisteix en traçar rajos des de la càmera i guardar-nos informació sobre el punt on impacten. Recordem que aquest pas només s'executarà quan calgui tornar a refer la imatge inicial, ja sigui perquè hem mogut la posició o orientació de la càmera o perquè s'ha canviat alguna configuració de l'algorisme que requereixi tornar a començar el refinament de la il·luminació.

8.1 Implementació de la càmera perspectiva

Aquest pas comença executant un programa de *Ray generation* que implementa una càmera perspectiva. Una càmera en perspectiva està definida pel punt on mira o *view reference point* (VRP), el qual apareixerà al centre de la imatge, la posició de l'observador on se situa la càmera (OBS), l'angle d'obertura vertical α i la relació d'aspecte r entre amplada i alçada del pla de projecció. A més, cal una base formada per tres vectors ortonormals que defineixen la seva orientació. Definim aquests tres vectors $\vec{u}$, $\vec{v}$ i $\vec{w}$ de la manera següent:

- $\vec{u}$ és el vector unitari en la direcció horitzontal del pla de projecció.
- $\vec{v}$ és el vector unitari en la direcció vertical del pla de projecció.
- $\vec{w}$ és el vector unitari en la direcció que uneix el VRP amb l'observador OBS.

A partir dels tres vectors anteriors, per a un píxel $(i, j) \in [0, n_w] \times [0, n_h]$ trobem la direcció el raig que travessa el seu centre fent:

$$(p_x, p_y) = \left(2\,\frac{i+0.5}{n_w} - 1,\ 2\,\frac{j+0.5}{n_h} - 1\right)$$

$$\vec{d} = s_w \cdot p_x \cdot \vec{u} + s_h \cdot p_y \cdot \vec{v} + \vec{w}$$

on $s_h = \text{dist}(\text{VRP}, \text{OBS}) \cdot \tan(\alpha/2)$ i $s_w = r \cdot s_h$. Fixem-nos que $\vec{d}$ no és unitari i ens caldrà normalitzar-lo per a construir el raig a OptiX, passant-li el punt on es situa l'observador i el vector unitari de direcció.

8.2 Traçat dels rajos

Els rajos primaris es llencen des del programa *Ray generation* amb la càmera que acabem d'explicar. Les estructures d'acceleració de l'escena són un SBVH per a cada edifici i un BVH per a les capses englobants de cadascun d'ells (veure secció 5.2.3). La intersecció

del raig amb l'escena es realitza a un programa de tipus *Intersection* on s'implementa el test d'intersecció entre un raig i un triangle. Com és molt comú, OptiX ja disposa d'una funció a la que li passem un raig i els tres vèrtexs del triangle i ens diu si hi ha intersecció i, en cas afirmatiu, la distància recorreguda pel raig, la normal al punt d'intersecció i les coordenades baricèntriques[1] del punt dins del triangle, que ens serviran per a interpolar els atributs dels vèrtexs.

Quan s'ha detectat una intersecció del raig amb l'escena, es cridarà al programa de *Closest hit* que li hem associat. Aquí definirem el comportament de les superfícies. Com s'ha vist a la secció 7.2.2, al canal *alpha* dels atles de textures de les façanes hem codificat si la superfície és especular. Per tant, accedirem a la textura, llegirem el color del *texel* corresponent i, si el canal *alpha* és zero considerarem la superfície com especular, altrament la considerarem difosa.

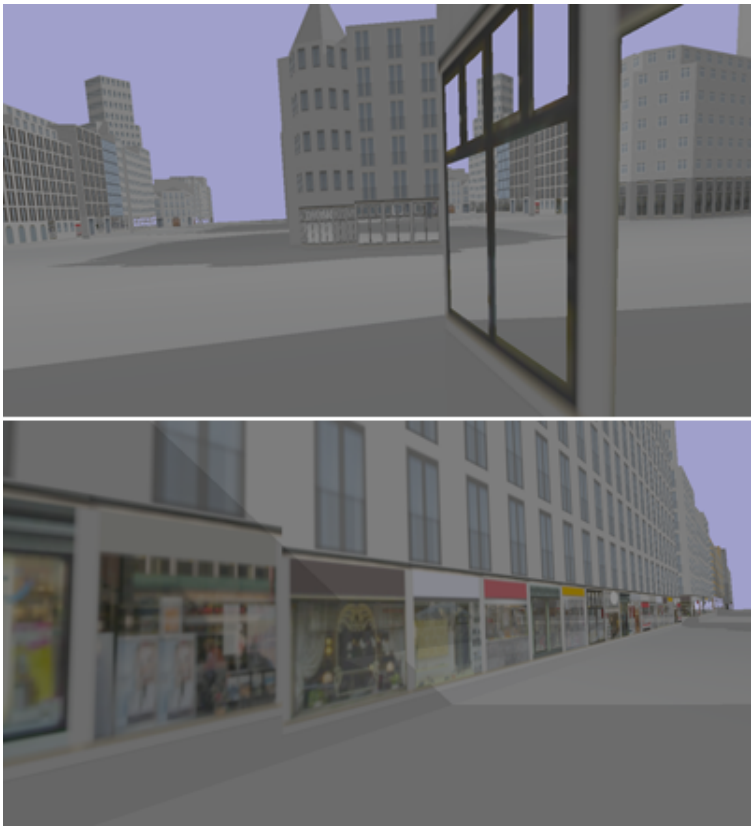

Figura 8.1: La imatge superior mostra el cas de reflexions on la reflectància és sempre 1. A la imatge de sota, s'ha aplicat l'aproximació de Schlick. Es pot veure com els aparadors que veiem més paral·lemament (cap a la dreta de la imatge) reflecteixen més que els que veiem més perpendicularment (cap a l'esquerra).

Si la superfície és difosa, senzillament prendrem el color dels canals RGB del *texel*. Omplirem l'estructura de dades que vam veure a la secció 4.4 quan vam descriure el Photon Mapping Progressiu, posant a la descripció de la BRDF aquest color multiplicat per l'atenuació del raig.

A les superfícies especulars calcularem el raig reflectit. La reflectància R_F d'aquest raig la calcularem amb l'aproximació de Schlick per als factors de Fresnel (veure secció

[1] Les coordenades baricèntriques (α, β, γ) d'un punt p dins d'un triangle d'àrea A amb vèrtexs t_1, t_2, t_3 es defineixen com: $\alpha = \text{area}(pt_2t_3)/A$, $\beta = \text{area}(t_1pt_3)/A$ i $\gamma = \text{area}(t_1t_2p)/A$. El punt p es pot expressar com $p = \alpha t_1 + \beta t_2 + \gamma t_3$, i si és interior al triangle es compleix $\alpha, \gamma, \beta > 0$ i $\alpha + \beta + \gamma = 1$.

2.2.3). Hem fixat el límit de reflexions especulars a 1, de manera que la següent superfície que intersequi es considerarà sempre com difosa. Fixem-nos que quan intersequi amb la superfície difosa, s'omplirà l'estructura de dades del píxel actual, i posteriorment li sumarem el color de la superfície on ha reflectit multiplicat per la transmitància $(1 - R_F)$. Això ho fem perquè hem codificat com especulars els vidres dels aparadors. D'aquesta manera, quan mirem una superfície molt perpendicularment, veurem el seu color, que equivaldria a veure a través del vidre de l'aparador i és el que conté la textura. Si la mirem paral·lelament, veurem les superfícies que s'han reflectit.

8.3 Accés a les textures

Per tal d'obtenir el color al punt d'una la superfície on ha intersecat el raig, cal saber si hem d'accedir a la ortofoto o als atles de textures. La solució és basar-se en la normal de la superfície, i aquelles que coincideixin amb la direcció vertical ens determinaran les superfícies on hem de llegir des de la ortofoto.

L'accés a la ortofoto, gràcies a com l'hem construït, és trivial. Com sabem que aquesta ocupa exactament la cara del pla del terra de la capsa de l'escena, les coordenades de textura c_s i c_t per accedir a la ortofoto són:

$$\begin{aligned} \text{texc}_s &= x/\text{capsa}_{\text{amplada}} \\ \text{texc}_t &= z/\text{capsa}_{\text{profunditat}} \end{aligned}$$

Per accedir als atles de textures a les façanes cal més informació referent a com localitzar la textura correcta dins l'atles de textures. Les dades que conté el model sobre la texturació, com hem vist a la secció 7.1.1 són el nom de la textura i les coordenades (s, t) de cada vèrtex, que estan expressades considerant que $(0, 0)$ és l'extrem inferior esquerre i $(1, 1)$ el superior dret. No obstant, ara les textures es troben agrupades als atles que hem construït com s'ha explicat a la secció 7.2. S'ha de definir l'accés correcte fent servir unes altres coordenades.

Recordem que teníem un total de 9 atles de textures, els 6 primers amb les textures reduïdes fins a una alçada de 32 píxels (amb amplada variables) i els altres 3 amb l'alçada reduïda a 64 píxels, ja que eren els comerços i volíem més detall. Per tant, cada atles ho podem veure com una matriu de textures. Per a localitzar una textura dins d'un atles ens farà falta saber a quina fila i columna d'aquest atles es troba. A més, ens caldrà saber l'amplada, ja que hem vist que pot ser variable. Per tant, ens fan falta 4 dades: l'identificador de l'atles, la fila, la columna i l'amplada de la textura.

En total ens cal codificar 6 variables, les 2 coordenades de textura i les 4 variables que localitzen la textura a l'atles. Afortunadament, els vèrtexs del model no tenen cap color assignat. Per tant, podem fer servir les tres components de color RGB i tres coordenades de textura STQ per a enviar la informació de texturació. Quan es carrega el model, es modifica el color i les coordenades de textura de cada vèrtex de la manera següent:

- `r` posició de l'extrem esquerre de la textura a l'atles, entre 0 i 1.
- `g` posició de l'extrem inferior de la textura a l'atles, entre 0 i 1.
- `b` amplada de la textura a l'atles, 1 voldria dir que ocupa tot l'ample de l'atles.
- `s` coordenada horitzontal de textura, es manté la del model original.
- `t` coordenada vertical de textura, es manté la del model original.
- `q` identificador de l'atles de textura, enter entre 0 i 8.

Com `r`, `g`, `b` i `q` només depenen de la textura, i cada cara només en tenia una assignada, tots els vèrtexs d'aquesta tindran el mateix valor i no ens hem de preocupar per la interpolació. Les coordenades `s` i `t` sí que seran interpolades. A més, com per a repetir la textura poden tenir valors superiors a 1, haurem de fer nosaltres aquesta repetició. Per últim, podem agrupar totes les textures a un *texture array* o una textura 3D (OptiX encara no té implementats els *texture array* a la versió 2.1 RC1). D'aquesta manera, l'accés a una textura concreta és més còmode.

Les coordenades finals amb les que accedirem a la textura des dels *shaders* GLSL o des del codi CUDA a OptiX les calculem com:

$$\begin{aligned} \text{texc}_s &= r + (s - \lfloor s \rfloor) \cdot b \\ \text{texc}_t &= g + (t - \lfloor t \rfloor) \cdot h \\ \text{texc}_q &= q + 0.5 \qquad \text{(a OptiX)} \\ \text{texc}_q &= (q + 0.5)/N \qquad \text{(a GLSL)} \end{aligned}$$

on N és el nombre total de atles de textures, que per a Barcelona és 9, i h és l'alçada de cada textura a l'atles, que per a Barcelona és $h = 1/32$ als sis primers atles i $h = 1/16$ als altres tres. La coordenada c_q ens és útil quan tenim tots els atles junts com a capes d'una sola textura 3D.

Figura 8.2: L'esquema de l'esquerra mostra un possible atles de textura i la interpretació dels valors que guardem al color del vèrtex. A la dreta s'ha visualitzat la ciutat amb el color assignat als vèrtexs.

Per últim, els rajos que no intersequin amb l'escena cridaran a l'execució del *Miss program*. Aquest llegirà una textura que farem servir per a mostrar el cel. Per a fer aquest accés es calcula a partir de la direcció unitària del raig $\vec{d} = (d_x, d_y, d_z)$ els angles $\theta = \arctan(d_x/d_z)$ i $\varphi = \frac{\pi}{2} - \arccos(d_y)$. A partir d'aquests, calculem les coordenades com:

$$\begin{aligned} \text{texc}_s &= (\theta + \pi)\frac{1}{2\pi} \\ \text{texc}_t &= \frac{1 + \sin\varphi}{2} \end{aligned}$$

La figura 8.3 mostra exemples de la imatge resultant de l'execució del pas de càmera, fent servir com a escena un model petit del centre de Barcelona.

Figura 8.3: Visualització directa (només el color, sense càlculs d'il·luminació) dels punts visibles obtinguts al pas de càmera.

Capítol 9

Pas de fotons

Seguint amb l'execució del Photon Mapping progressiu, el segon pas és el traçat de fotons i guardar-los al *Photon Map*. Els fotons s'originaran a l'única font de llum de l'escena, que és el Sol, es propagaran per l'escena permetent un nombre màxim de rebots i anirem emmagatzemant informació sobre aquestes interseccions amb la geometria al mapa de fotons.

9.1 Traçat dels fotons

El traçat de fotons és bastant similar al traçat de rajos que hem descrit al capítol anterior. A cada pas generarem $N_w \times N_h$ fotons, on N_w i N_h són, respectivament, l'amplada i alçada d'un *buffer* que inicialitzarem amb valors aleatoris. Iniciarem el programa de *Ray generation* associat al pas de fotons i, per cada posició del *buffer*, construirem el raig primari del fotó.

Com el Sol el podem considerar una font de llum direccional a causa de la distància a la Terra, aquesta vegada la direcció és trivial d'obtenir. Per a major comoditat, el programa permet canviar la direcció del Sol en coordenades polars (r, φ, θ) on $\varphi \in [0, 2\pi]$ i $\theta \in [-\pi/2, \pi/2]$. El vector director és $\vec{d} = (d_x, d_y, d_z) = (\cos\theta\cos\varphi, \sin\theta, \cos\theta\sin\varphi)$.

El que variarem a cada raig és la posició d'origen. Calcularem el rectangle englobant de la projecció de l'escena al pla que té per normal la direcció de la llum i el discretitzarem en N_w i N_h cel·les. La cel·la central coincidirà amb la posició en coordenades polars del Sol, que és $r \cdot \vec{d}$, i la posició de la resta la calculem de manera similar a com implementàvem la càmera perspectiva:

$$(p_x, p_y) = \left(2\,\frac{i+0.5}{n_w} - 1,\ 2\,\frac{j+0.5}{n_h} - 1\right)$$

$$\vec{p} = 0.5 \cdot s_w \cdot p_x \cdot \vec{u} + 0.5 \cdot s_h \cdot p_y \cdot \vec{v} + r \cdot \vec{d}$$

on $\vec{u}$ i *vecv* són els vectors unitaris en la direcció horitzontal i vertical del rectangle englobant de la projecció, i s_w i s_h són les mides horitzontal i vertical d'aquest, respectivament.

Si sempre fem servir l'equació anterior per a calcular la posició $\vec{p}$, els fotons sempre impactaran per primer cop contra la mateixa superfície. Per tant, afegim una pertorbació aleatòria movent $\vec{p}$ a qualsevol de les posicions contingudes dins la cel·la.

Quan un fotó impacta sobre una superfície especular, calcularem la nova direcció de propagació com el raig reflectit. Si arriba a una superfície difosa, la nova direcció de propagació serà calculada aleatòriament d'entre totes les possibles a l'hemisferi sobre el punt d'intersecció. A més, quan la superfície sigui difosa, guardarem la informació del punt d'impacte al mapa de fotons *si no és la primera interacció del fotó amb l'escena*, ja que aleshores seria llum directa i aquesta la calcularem posteriorment.

Inicialment, els fotons començaran amb un flux que serà el mateix per a totes les components RGB. A cada interacció amb l'escena llegirem les textures de façanes que hem generat al preprocés (secció 7.4) i multiplicarem cada component per la del color que llegim d'aquesta textura. D'aquesta manera, si una superfície fos vermella, per exemple, el flux del fotó quan se segueixi propagant tindrà la component vermella major que les altres. Així podem aconseguir la difusió del color o *color bleeding*.

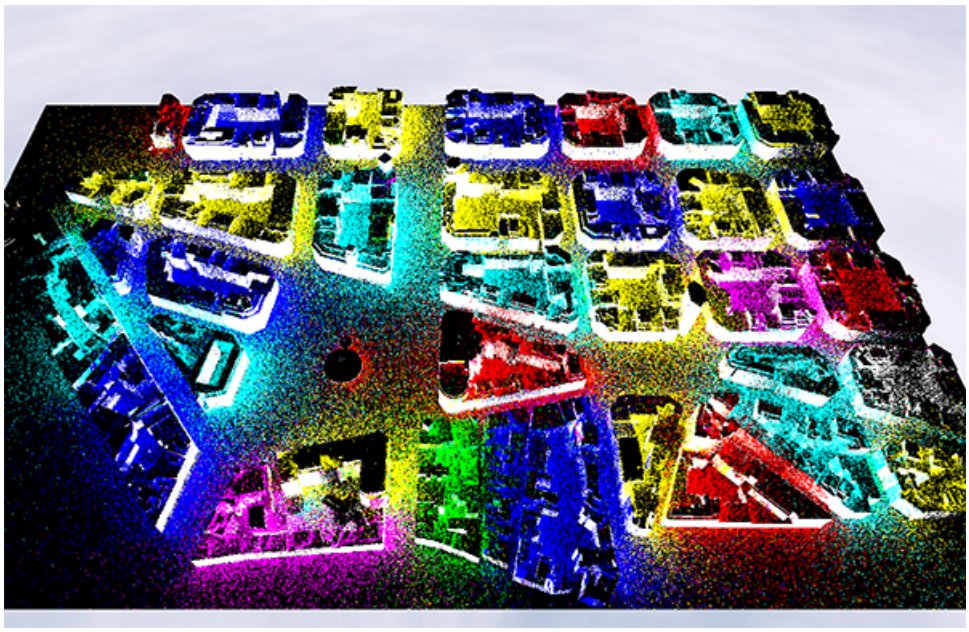

Figura 9.1: Visualització dels impactes dels fotons sobre l'escena. El Sol està situat a la dreta ($\varphi = 0$) i elevat 45° ($\theta = \pi/4$). Per a fer més clara la imatge, a cada edifici s'ha assignat un color i al terra el color blanc. Es pot observar com el terra queda rep el color dels edificis on els fotons han rebotat, mentre que els edificis queden majoritàriament tenyits de blanc degut a que els fotons que hi arriben provenen gairebé sempre del terra. Observem també que els terrats són gairebé tot negres perquè que reben poca il·luminació indirecta.

9.2 Estructura del mapa de fotons

Com hem vist al fer la descripció general de l'algorisme, la principal aportació que fem és la representació del mapa de fotons. Si analitzem la geometria d'una ciutat, veiem que gairebé en tots els casos podem classificar els polígons en aquells que són paral·lels al pla del terra (el propi terra i terrats d'edificis) i aquells que són perpendiculars (les façanes). Per tant, podem buscar alguna manera d'emmagatzemar-los que realment sigui especialitzada per a aquest tipus d'escena.

Un altre detall important dels models urbans és que gairebé mai trobarem una paret que puja allunyant-se de la vertical de l'edifici cap enfora. O una que puja perpendicu-

larment des del terra, avança cap fora de l'edifici i després segueix pujant. Per tant, una representació molt acurada de totes les superfícies horitzontals és una ortofoto de l'escena. Per tant, podem emmagatzemar els impactes dels fotons sobre superfícies horitzontals a una **textura amb la projecció ortogonal** de l'escena sobre el pla del terra. Un altre avantatge és que l'accés a la posició concreta dins d'aquesta textura a partir de la posició del punt d'impacte és trivial, es fa de la mateixa manera que quan volíem llegir el *texel* de la ortofoto per agafar el color: $s = x/\text{capsa}_{\text{amplada}}$, $t = z/\text{capsa}_{\text{profunditat}}$.

Pel que fa a les parets verticals dels edificis, la majoria d'ells es poden classificar en dos tipus: una illa de cases amb pati interior o un bloc o illa de cases sense pati. A més, una característica que també trobem gairebé sempre és que, per a un mateix edifici, dues façanes orientades en el mateix sentit (cap al carrer o cap al pati interior si n'hi ha) no s'obstrueixen la visibilitat entre elles. Per tant, si projectem les façanes exteriors sobre un cilindre vertical la representació d'aquestes serà acurada, de la mateixa manera que si ho fem amb les façanes interiors.

Per tant, per a les façanes farem la distinció entre exteriors i interiors. Per a saber en quin tipus de façana ens trobem, ens farà falta saber l'identificador de l'edifici on ha impactat el fotó i la normal al punt d'impacte, dades que les calcularà el programa que implementa el test d'intersecció del raig amb la geometria. Si tenim un vector amb els centres de la capsa englobant de cada edifici, només ens farà falta fer el producte escalar de la normal al punt d'intersecció amb el vector que uneix el centre amb aquest punt. Si el resultat és positiu, serà una façana exterior, altrament serà una façana interior. En base a aquesta decisió guardarem el fotó a la **textura de façanes exteriors** o a la **textura de façanes interiors**. Per a trobar la posició dins d'aquesta textura farem servir la parametrització cilíndrica descrita a la secció 7.4.1.

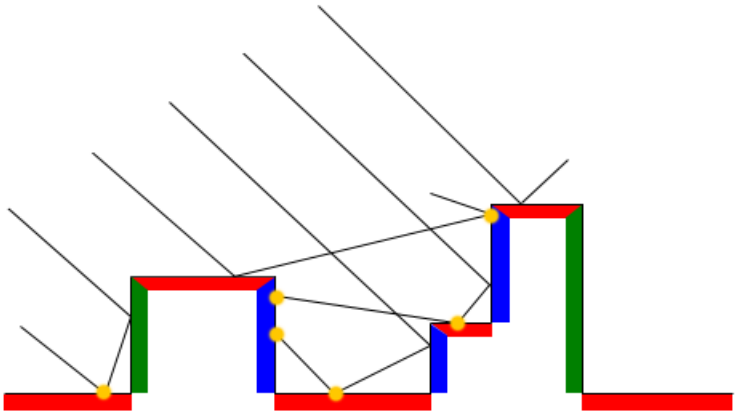

Figura 9.2: Classificació dels tipus de superfície on pot impactar un fotó. L'esquema mostra una secció vertical d'una illa de cases amb pati interior. En vermell es mostren aquelles superfícies representades al mapa de terres. En verd les que corresponen a façanes exteriors i en blau les de façanes interiors. Els cercles grocs representen els punts on impacten els fotons que guardarem. Observem com la primera interacció, que és llum directa, no la guardarem.

En resum, el nostre mapa de fotons està format per tres textures. Si la normal del punt d'impacte d'un fotó és gairebé la direcció vertical, es guardarà a la textura de terres. Altrament, en funció de si la superfície està orientada cap al carrer o al pati interior (en cas d'haver-hi) guardarem el fotó a la textura de façanes exteriors o interiors. Aquestes textures es faran servir al següent pas per a calcular la il·luminació indirecta i seran borrades posteriorment, començant sempre buides cada vegada que es fa el pas de fotons.

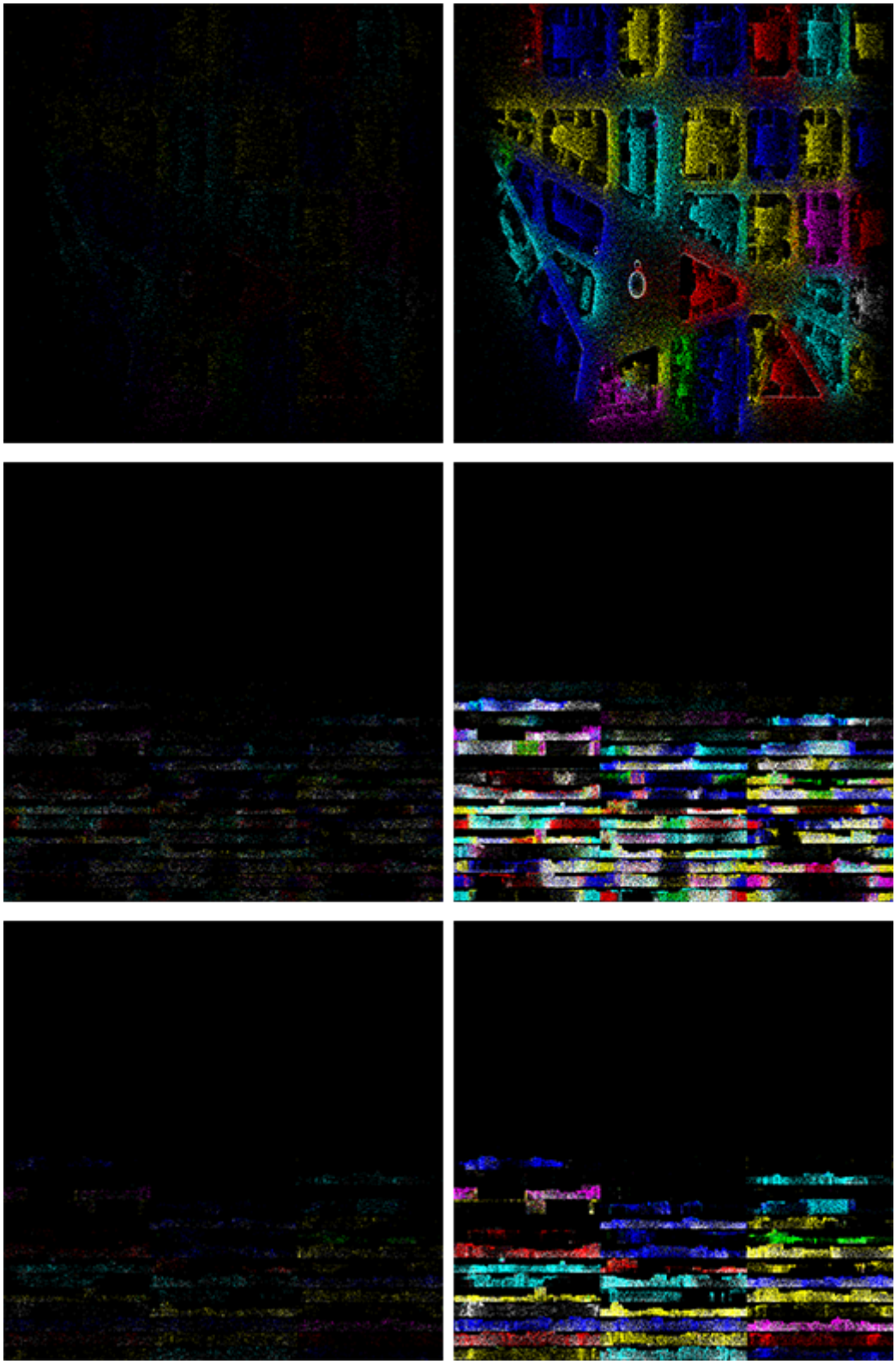

Figura 9.3: De dalt a baix les tres textures que representen el nostre mapa de fotons: textura de terres, textura de façanes exteriors i textura de façanes interiors. La imatge de l'esquerra representa la textura per a una iteració del pas de fotons. La de la dreta, el resultat acumulat de 20 iteracions.

Capítol 10

Pas d'il·luminació

El resultat del pas de càmera és un *buffer* amb informació sobre les superfícies visibles des de la càmera, és a dir, la informació sobre els materials. El resultat del pas de fotons són tres textures que representen la distribució dels impactes de fotons sobre l'escena, és a dir, la informació sobre la llum. El que cal fer a continuació és combinar aquestes dades per a calcular la il·luminació de l'escena i obtenir la imatge resultant. Per a cada posició del *buffer* resultat del pas de càmera, calcularem el valor de la il·luminació al punt de l'escena x corresponent fent la suma de tres components:

- L_d: la il·luminació directa des de les fonts de llum.
- L_i: la il·luminació indirecta, representada pel *Photon map*.
- L_a: la il·luminació ambient.

10.1 Il·luminació directa i ambient

La il·luminació directa l'obtindrem directament des de les fonts de llum, ja que com hem vist al capítol anterior el primer impacte dels fotons no es registra al *Photon map*. El que farem serà comprovar abans si la llum directa arriba al punt que volem il·luminar.

Com la llum és direccional, només fa falta crear un raig que tingui com a punt d'origen el propi punt de la superfície pel qual estem fent aquest test i com a direcció la de la llum en sentit invers. Aquest tipus de raig tindrà associat un programa de *Any hit*, de manera que si interseca amb alguna superfície, indistintament de quina sigui, sabem que no li arriba llum directa i la contribució serà nul·la. Una petita optimització que podem fer és guardar-nos després del primer *frame* el resultat d'aquest test per a cada posició del *buffer* de càmera, així només haurem de tornar a fer-lo quan canviï la càmera.

Si la llum directa arriba al punt, calcularem el terme d'il·luminació directa L_d a partir de la potència P de la llum, la distància d entre la llum i la superfície i la direcció unitària $\vec{l}$ de la llum com:

$$L_d = P \cdot k_d \cdot \vec{n}_x \cdot \vec{l}$$

on $\vec{n}_x$ és la normal al punt que estem il·luminant i k_d la component difosa de la seva BRDF.

El terme d'il·luminació ambient L_a senzillament el sumarem al càlcul final d'il·luminació. Consisteix en un coeficient $a \in [0, 1]$ habitualment molt baix, com ara $a = 0.05$ multiplicat per la component difosa de la BRDF. És una manera de representar empíricament una

part de la il·luminació indirecta que arriba a totes les superfícies per igual i que costaria traçar molts fotons per aconseguir el mateix efecte.

$$L_a = a \cdot k_d$$

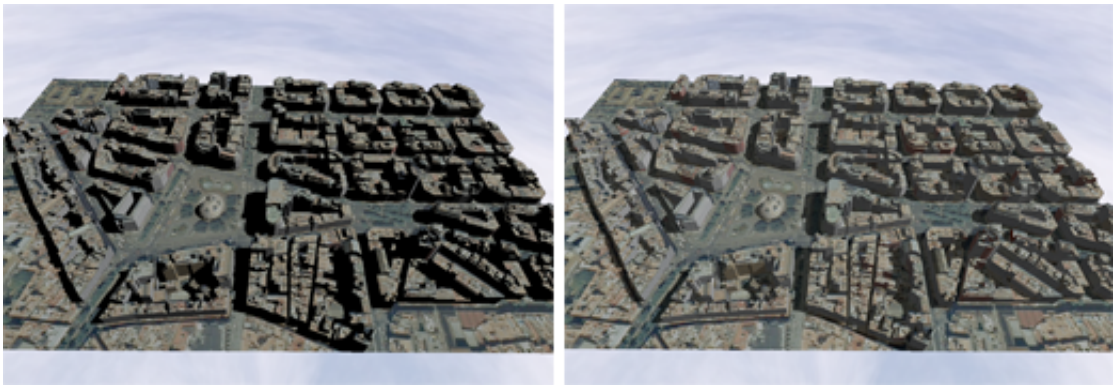

Figura 10.1: Il·luminació amb llum només directa (esquerra) i amb llum directa i ambient fent servir un coeficient $a = 0.05$ (dreta).

10.2 Il·luminació indirecta

La llum indirecta, que haurà rebotat com a mínim un cop a les superfícies difoses, està representada als mapes de fotons. D'aquests podrem obtenir el flux acumulat Φ i el nombre N de fotons que hi ha al voltant del punt x amb un radi r i calcular un estimador similar al que es descriu a la secció 4.3.

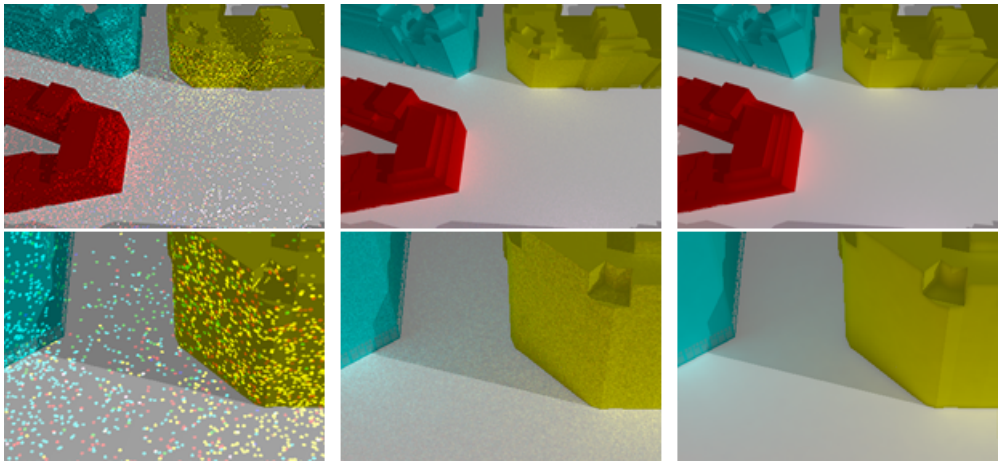

Figura 10.2: Comparació dels resultats obtinguts amb diferents radis utilitzats per a buscar els fotons. A l'esquerra es mostren els fotons directament, sense estimar l'àrea. Al centre, $r^2 = 0.01$. A la dreta, $r^2 = 8.0$.

La posició del punt i la normal venen donades pel *buffer* de la càmera. Recordem que vam veure a la versió original del Photon Mapping que teníem diverses dades guardades per cada node del *kd-tree*, com la posició i la normal de la superfície, la direcció d'incidència del fotó i el flux. Amb la nostra estructura del mapa de fotons aquestes dades ja venen implícites. La única que no podrem obtenir és la direcció d'incidència del fotó, però com

es fa servir per a avaluar la BRDF i multiplicar-la pel flux, guardem als mapes de fotons el flux ja multiplicat per aquesta direcció.

El primer que s'ha de fer es diferenciar si la superfície és un terra o una façana. Com tenim la normal $\vec{n}_x$ també guardada al *buffer* de càmera, podem establir el criteri que les normals properes a la vertical seran terres, i la resta correspondran a façanes. Un cop sabem quin tipus de superfície és, agafem els fotons de les textures apropiades. Ens guardarem el nombre de fotons N que trobem i la suma dels seus fluxos Φ_i. L'estimador de la radiància per a la il·luminació indirecta el calculem com:

$$L_d = \frac{1}{\pi r^2} \sum_{i=1}^{N} \Phi_i$$

Recordem també de la secció 4.4 que anirem disminuint progressivament el radi amb un factor de reducció basat en la quantitat de fotons que ja teníem acumulats i els nous que han arribat.

10.2.1 Fotons del terra i els terrats

El mapa de fotons de terres i terrats (farem referència d'ara endavant a tots dos com a terres) és una discretització de la projecció sobre el pla horitzontal en $T_w \times T_h$ píxels. Dit d'una altra manera, un píxel del mapa de fotons equival a un rectangle sobre el terra que tindrà una amplada $w_{\text{pixel}} = \text{capsa}_{\text{amplada}}/T_w$ i alçada $h_{\text{pixel}} = \text{capsa}_{\text{profunditat}}/T_h$, on capsa és la capsa englobant de l'escena. Per tant, si volem agafar els fotons dins d'un radi r, podem aproximar-lo com els fotons que es trobin dins d'un rectangle de $(2\lceil r/w_{\text{pixel}}\rceil + 1) \times (2\lceil r/h_{\text{pixel}}\rceil + 1)$ píxels, i el píxel central del rectangle és el píxel on es troba el punt x de la superfície que estem il·luminant.

Quan estiguem mirant els fotons d'aquest rectangle haurem de tenir en compte que podrien estar a una alçada diferent a la de x. Per exemple, si x es troba al terra del carrer prop d'una paret, és possible que estiguem incloent píxels que en realitat formen part del terrat que hi ha dalt de la paret. Per a solucionar aquest tipus de situacions, vam crear al prepocés un mapa d'alçades dels edificis (veure secció 7.3.2). Si l'alçada a la que es troba el píxel que estem mirant no és la mateixa que la del píxel corresponent a x, el descartarem i no acumularem els seu flux.

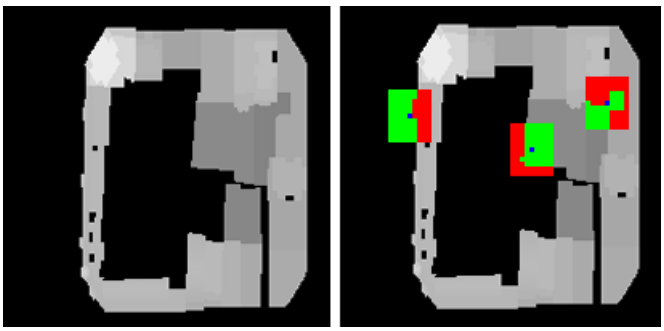

Figura 10.3: La imatge de la dreta mostra superposat al mapa de profunditats de l'esquerra els rectangles d'on obtenim els fotons. El quadrat blau seria el píxel central des d'on iniciem la cerca. Els píxels verds són els que llegirem del mapa de fotons, i els vermell els que descartem.

10.2.2 Fotons de les façanes

De manera similar al cas dels terres, el mapes de façanes contenen el conjunt de façanes dels diversos edificis. Per determinar si fer servir el mapa de façanes exteriors o interiors mirarem el signe del producte escalar entre la normal de la superfície $\vec{n}_x$ i el vector que uneix el centre de la capsa englobant de l'edifici amb el punt x. Si és positiu serà una façana exterior, i si és negatiu serà interior.

Com cadascun dels edificis ocupa $F_w \times F_h$ píxels dins d'aquests mapes, podem saber cada píxel a quin rectangle projectat sobre la façana correspon. L'amplada w_{pixel} la considerarem una fracció del perímetre de la circumferència que fa de base del cilindre. Si sabem el radi R_c d'aquesta, $w_{\text{pixel}} = 2\pi R_c/F_w$. Per a calcular l'alçada h_{pixel} necessitarem saber l'alçada de l'edifici, de manera que $h_{\text{pixel}} = \text{edifici}_{\text{alçada}}/F_h$. Finalment, el rectangle de píxels del mapa de façanes on buscarem els fotons dins un radi r serà $(2\lceil r/w_{\text{pixel}}\rceil+1)\times(2\lceil r/h_{\text{pixel}}\rceil+1)$, com teníem al cas anterior. La posició del píxel central la trobem aplicant la parametrització cilíndrica de l'edifici i el correcte posicionament dins la regió concreta de la textura.

Si el rectangle inclou files de píxels per sota dels que estan assignats a l'edifici dins la textura, vol dir que hem baixat per sota l'alçada mínima y_{min} d'aquest edifici i podem descartar directament aquestes files. En canvi, si ens sortim per l'esquerra o per la dreta de la regió de píxels assignada vol dir que estem creuant la recta de tall del cilindre, i el que farem serà continuar prenent columnes de píxels a l'altre costat.

Finalment, amb les façanes haurem també de fer servir el mapa de profunditats descrit a la secció 7.4. Per exemple, gairebé sempre la parametrització d'una illa de cases ens dóna un perfil d'alçades variables. Per tant, cal saber si el rectangle ha sortit per sobre l'alçada de l'edifici. De manera similar, a vegades els blocs de pisos són esglaonats i alguns pisos tenen profunditat diferent dels de sota, deixant un terrat o replà entre els diferents nivells. Anàlogament, el bloc de pisos veí podria estar situat a una profunditat considerablement diferent de l'actual, i no hauríem de fer que els fotons de la façana d'un contribueixin a la il·luminació de la façana de l'altre.

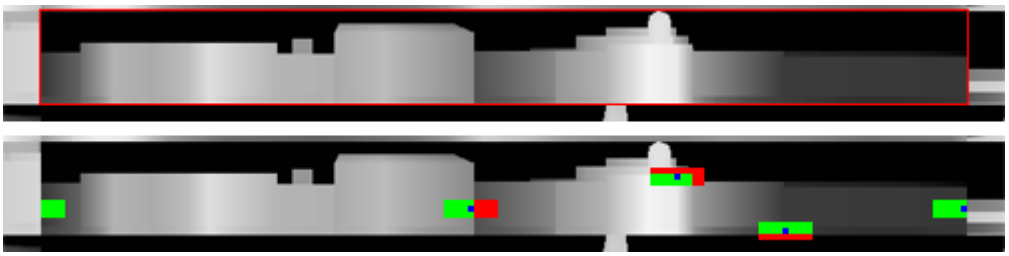

Figura 10.4: La imatge de la dreta mostra superposat al mapa de profunditats de l'esquerra els rectangles d'on obtenim els fotons. El quadrat blau seria el píxel central des d'on iniciem la cerca. Els píxels verds són els que llegirem del mapa de fotons, i els vermell els que descartem. Fixem-nos en els casos concrets de les façanes corresponents a quan baixem per sota el píxel mínim i quan ens cal donar la volta per l'altre costat del desplegament.

10.2.3 Combinació de fotons entre terres i façanes

Si tractem de manera completament separada els fotons dels terres i els de les façanes, a les interseccions entre terra i paret l'estimador donarà un resultat més fosc. El motiu és

que part de l'àrea dins la qual busquem els fotons es troba "dins" de la paret (o del terra) i la densitat de fotons serà menor. Una possible solució és detectar aquests casos i fer que es mirin també els fotons de la façana/terra en cada cas. Com l'àrea que representa un pixel de les façanes i un píxel del terra no té per què ser equivalent, escalarem el flux dels fotons que agafem segons la relació que hi ha entre l'àrea dels píxels actuals i la dels de l'altre mapa. Si no ho fem, quedaran zones bastant fosques i zones massa brillants.

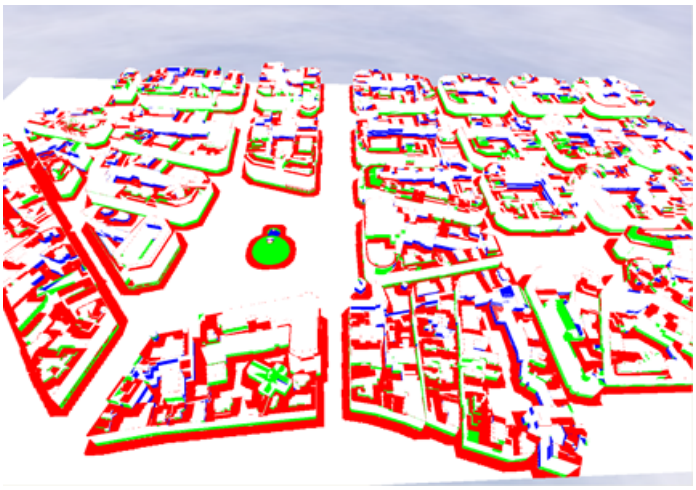

Figura 10.5: Combinació de fotons entre façanes i terres. En vermell es mostren els punts de terra que prenen fotons també de les façanes. En verd, les façanes exteriors que agafen fotons del terra i, en blau, les interiors.

Cas terra → façana

Quan estem agafant fotons de terra i detectem amb el mapa d'alçades que estem a una alçada diferent, vol dir que hi ha hagut alguna paret. Tenim dos casos possibles: que l'alçada sigui menor a la inicial, o que sigui major:

- Si l'alçada és menor, vol dir que estàvem en algun terrat i hem trobat una paret cap avall fins a un altre terrat inferior o el terra. Per tant, tenim una intersecció convexa i els fotons d'aquesta paret que baixa no formen part de l'hemisferi sobre el punt x.
- En canvi, si l'alçada és major i, per tant, la paret ha pujat, tenim una intersecció còncava i els fotons de la paret voldrem que afectin a la il·luminació del punt del terra, ja que es troben dins l'hemisferi d'aquest.

Per a saber la posició dins el mapa de fotons de façanes ens cal saber quin edifici és. Fins ara aquest problema s'havia solucionat gràcies a que guardàvem l'identificador de l'edifici on impactava el raig del pas de càmera. Però ara l'identificador és el que correspon al terra, així que cal buscar una altra solució per trobar l'edifici. La textura d'on llegim el mapa d'alçades, com s'explica a la secció 7.3.2, conté als canals de color RGB codificat l'identificador de l'edifici. Si hem detectat un canvi d'alçada, com el terra és un pla, segur que serà perquè estem sobre un edifici. Per tant, recuperem l'identificador fent servir aquesta textura d'identificadors.

Un cop sabem l'identificador, hem de calcular la posició i la normal del punt actual p per a poder calcular la seva projecció cilíndrica i llegir el mapa de fotons de façanes. Les coordenades p_x i p_z les trobem basant-nos en la posició inicial x i en la mida de cada píxel de la textura de fotons de terra. L'alçada l'aproximem amb la distància a x sumada a l'alçada a la que es trobava x. La normal $\vec{n}_p$, degut a que totes les interseccions entre

terra i paret que tractem són còncaves, apuntarà cap a x. Com només es fa servir per a determinar si estem en una façana exterior o interior, no ens cal més precisió. Per tant:

$$p = (x_x + d_x,\ x_y + \sqrt{d_x^2 + d_z^2},\ x_z + d_z)$$

$$\vec{n}_p = (-d_x,\ 0,\ -d_z)$$

on $d_x = w_{\text{pixel}} \cdot \Delta\text{columnes}$ i $d_z = h_{\text{pixel}} \cdot \Delta\text{files}$, prenent w i h com les mides del píxel del mapa de fotons.

Cas façana → terra

Afortunadament, passar al terra quan estem sobre una façana és més senzill que el pas invers que acabem de veure. Com abans, només ens interessarem per aquelles interseccions còncaves. Hi ha dos casos que hem de detectar:

- El primer d'ells és quan hauríem d'accedir a files de píxels per sota la fila més baixa assignada per a la parametrització de l'edifici. Això vol dir que hem arribat a un terra.
- L'altre cas és quan la profunditat d'un píxel que es troba per sota el píxel central associat al punt x és major que la d'aquest. Això vol dir que la façana està més cap enfora, ja sigui cap al carrer en façanes exteriors o cap al centre en façanes interiors, d'acord amb com hem generat el mapa de profunditats de les façanes. Per tant, hi ha un terrat o replà entre aquestes dues façanes que fa intersecció còncava amb la façana on es troba x. No haurem d'agafar més files de píxels del mapa de façanes, ja que s'ha trencat la continuïtat, i els haurem d'agafar del terra.

Per a calcular p_x i p_z (no ens cal l'alçada p_y per accedir al mapa dels terres) suposarem que la direcció $\vec{f}$ de la façana on es troba el punt x associat al píxel central és perpendicular a la normal $\vec{n}_x$ sobre el pla del terra. Per tant, $\vec{f} = (0,1,0) \times \vec{n}_x = (n_{x,z}, 0, -n_{x,x})$. El punt concret sobre el terra que farem servir per accedir al mapa de fotons de terra serà:

$$p = x + d_x\vec{n}_x + d_z\vec{f}$$

on $d_x = w_{\text{pixel}} \cdot \Delta\text{columnes}$ i $d_z = h_{\text{pixel}} \cdot \Delta\text{files}$ com ja hem vist abans, però ara w_{pixel} i h_{pixel} són les mides dels píxels dels mapes de façanes.

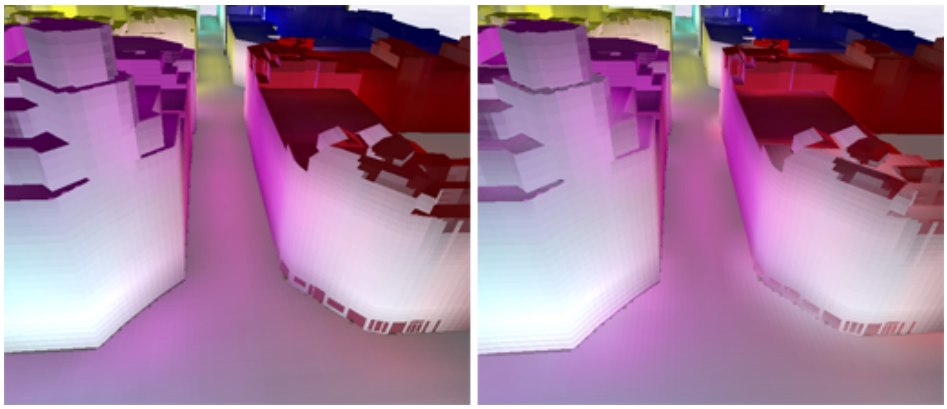

Figura 10.6: La imatge de la dreta mostra la mateixa escena amb les mateixes condicions d'il·luminació que la de l'esquerra, però combinant els fotons entre façanes i terres. Només es mostra la il·luminació indirecta, amb edificis colorejats i exagerada, per a fer més clar l'efecte.

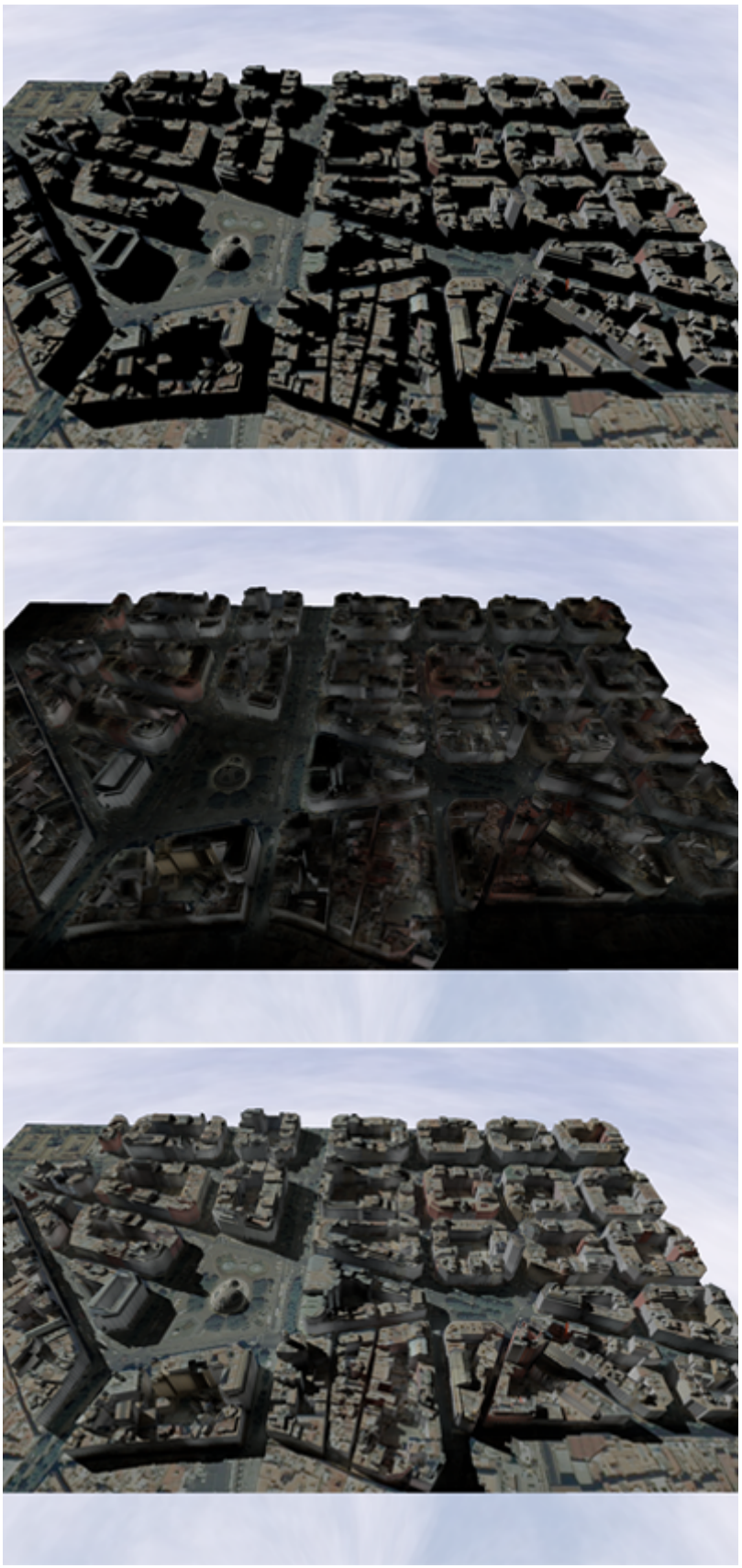

Figura 10.7: Resultat final del pas d'il·luminació (a baix) després de sumar la component directa (a dalt) i la component indirecta (al centre).

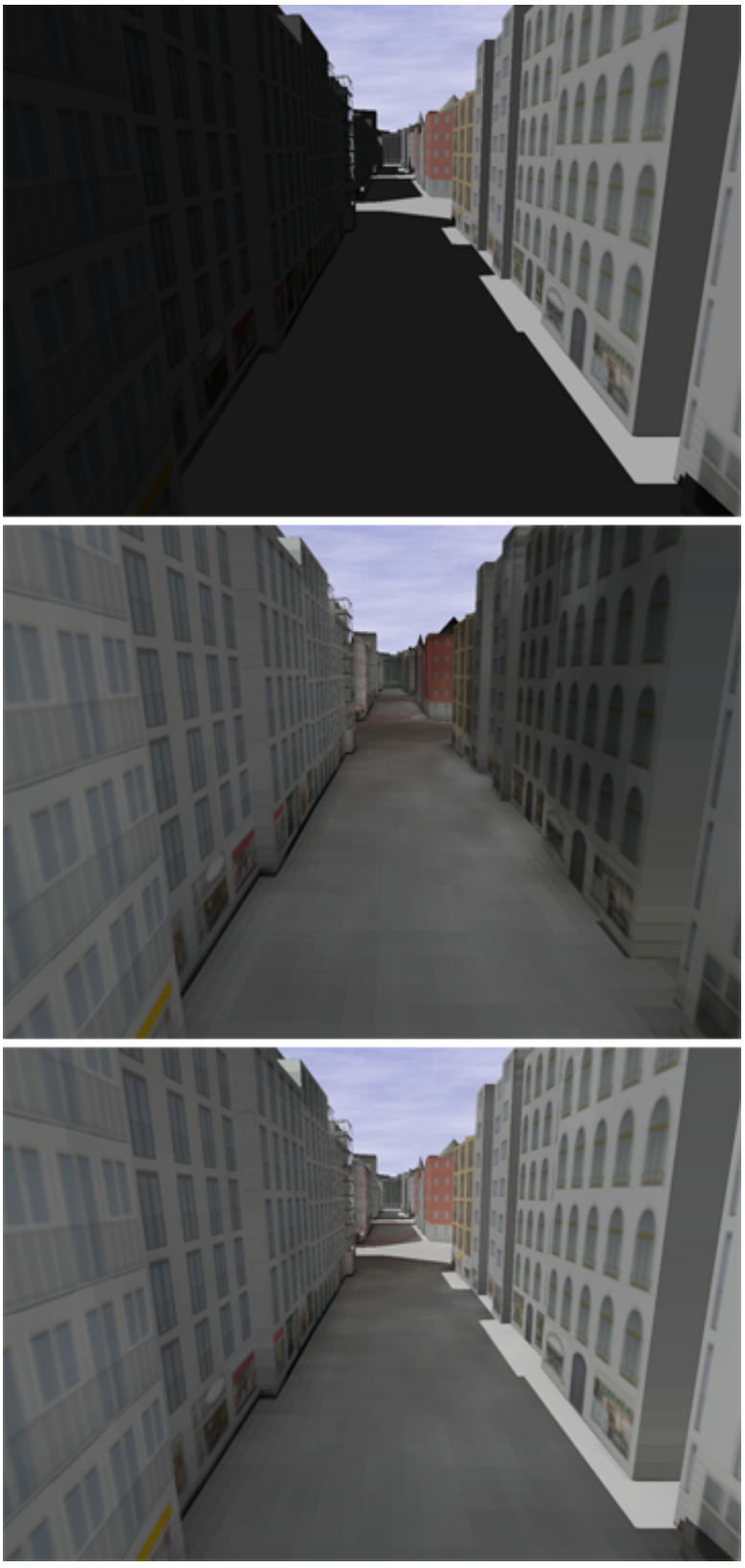

Figura 10.8: Resultat final del pas d'il·luminació (a baix) després de sumar la component directa amb un coeficient ambient de 0.01 (a dalt) i la component indirecta (al centre).

Capítol 11

Extensions del mapa de fotons

Als capítols anteriors hem vist l'algorisme que hem desenvolupat per a aplicar Photon Mapping a models d'entorns urbans. El mapa de fotons que fèiem servir era una textura RGBA. A continuació, es presenten dues variants o extensions de l'algorisme que permeten aconseguir millors visualitzacions augmentant la resolució del mapa de fotons sense haver de gastar més memòria. La figura 11.1 compara les diferents resolucions que obtenim amb les dues extensions que s'expliquen a continuació.

Figura 11.1: Comparació de la mida dels fotons segons la variant, a una escena amb 44 edificis. A l'esquerra, el mapa de fotons és en color i cada edifici està parametritzat en 340×34 píxels. Al centre, corresponent al mapa de luminàncies, els edificis es parametritzen sobre 680×68 píxels. A la dreta, l'edifici es visualitza amb el major nivell de detall dels mapes adaptatius, i correspon a 1024×102 pixels.

11.1 Mapes de luminància dels fotons

Una proposta molt senzilla és utilitzar textures d'un sol canal per a emmagatzemar els fotons. D'aquesta manera, no estarem guardant el color sinó només on han impactat fotons. Al passar de 4 canals a només un podem fer una textura el doble d'ample i el doble d'alta que ocuparà el mateix en memòria. L'àrea d'un píxel d'aquestes textures serà una quarta part de l'àrea que teníem amb les de color, i per tant tenim molta més precisió a les posicions dels fotons i la il·luminació indirecta queda més suau.

L'inconvenient d'aquest tipus de mapa de fotons és que perdem el *color bleeding*, però almenys amb el model de Barcelona i les textures de les que disposàvem no era un efecte massa significatiu. Tot i això, podem seguir fent el traçat de fotons amb les tres components RGB de color, i multiplicar el flux a cada component pel coeficient difós corresponent, i quan calgui desar el fotó al mapa fer algun tipus de conversió. Fem dues propostes:

- Escollir una de les components, per exemple la de valor més baix o més alt, i desar només aquesta.

- Convertir el color d'espai RGB a espai YUV. L'espai de color YUV representa un color mitjançant tres components: la luminància o brillantor general de la imatge (Y) i dos canals de *croma* (U, V) que corresponen a la diferència escalada de la luminància amb dues de les components de color en RGB. Com només ens interessa la luminància, calculem:

$$Y = 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B$$

  Els pesos de cada component poden ser uns altres, però normalment es dóna més pes al verd, ja que és el color pel qual els humans percebem més la brillantor, i menys al blau.

L'opció que hem implementat ha estat quedar-nos amb la luminància. Quan vulguem llegir els fotons, per no variar tota la resta de l'algorisme que ja havíem implementat, farem una conversió a escala de grisos: replicarem el valor de Y emmagatzemat al mapa de fotons a les tres components RGB del color del fotó.

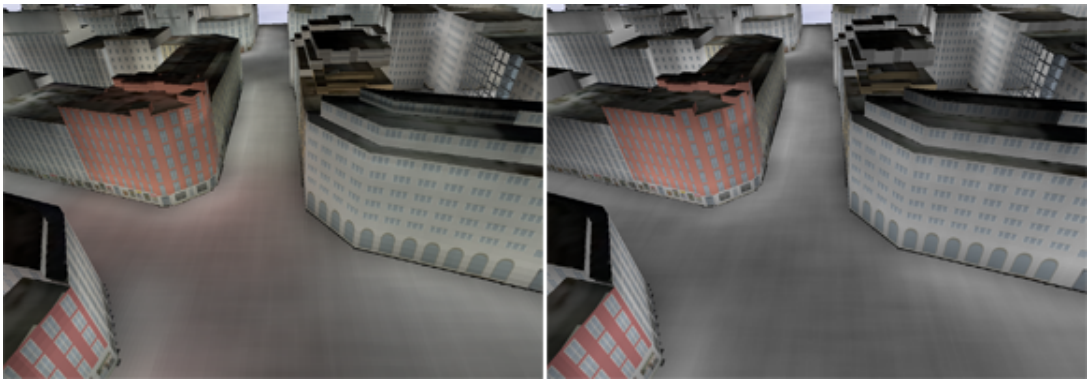

Figura 11.2: La imatge de l'esquerra mostra una de les zones on es fa més visible el *color bleeding*. La imatge de la dreta mostra la mateixa zona amb només els valors de luminància. Es mostra només la il·luminació indirecta i amb el terra sense texturar per poder apreciar millor el canvi.

11.2 Mapes de fotons adaptatius

L'altra proposta que fem consisteix en centrar-se en obtenir el màxim detall possible als fotons dels edificis que estem veient a prop, i anar-la disminuint a mesura que ens allunyem.

Si volem fer servir la mateixa memòria que amb la implementació de mapes de fotons en color, tindrem 4 textures per mapa de fotons d'un sol canal per a la luminància, com a la secció anterior. Ara, però, aquestes 4 textures seran de la mateixa mida que la textura de fotons en color.

De la mateixa manera que vam deduir a la secció 7.4.2 l'expressió per a col·locar tots els edificis a la textura de manera òptima, ara ens caldrà refer aquest càlcul per a que quedin repartits entre les 4 textures fent servir diferents nivells de detall (LODs). Voldrem que a la textura associada al millor nivell de detall tinguem els edificis el més grans possible. A més, per a simplicitat de l'algorisme, farem que l'amplada i alçada dels edificis es redueixi a la meitat cada vegada que passem al següent nivell de detall. Per tant, si tenim N edificis:

$$\begin{aligned} N &\leq N_{\mathrm{LOD}_0} + N_{\mathrm{LOD}_1} + N_{\mathrm{LOD}_2} + N_{\mathrm{LOD}_3} \\ &= N_{\mathrm{LOD}_0} + 4N_{\mathrm{LOD}_0} + 16N_{\mathrm{LOD}_0} + 64N_{\mathrm{LOD}_0} \\ &= 85N_{\mathrm{LOD}_0} \end{aligned}$$

Per tant, al nivell 0, que serà on els edificis siguin més grans, han de caber $\lceil N/85 \rceil$ edificis. Combinant-ho amb la relació 10:1 que teníem entre l'ample i l'alt dels edificis podem calcular el nombre c de columnes que tindrà la textura del nivell 0 com:

$$c = \left\lceil \sqrt{N/850} \right\rceil$$

Un cop sabem quants edificis han d'anar a cada nivell només cal anar-los omplint començant pel de més resolució. Per tal de decidir quins edificis volem a cada nivell els assignarem una puntuació a cadascun. El factor de puntuació p el calculem basant-nos en la distància entre l'edifici i l'observador, i en la posició que tindrà a la pantalla. Així, donem més prioritat a aquells edificis propers i que estiguin centrats a la pantalla. Si tenim la distància entre l'observador i el centre de la capsa englobant de l'edifici, així com els vectors unitaris al pla del terra $\vec{v}_{\mathrm{cam}}$, corresponent a la direcció de visió de la càmera, i $\vec{v}_{\mathrm{centre}}$, per a la direcció que uneix l'observador amb el centre de la capsa, calculem:

$$p = \frac{dist}{\vec{v}_{\mathrm{cam}} \cdot \vec{v}_{\mathrm{centre}}}$$

Els edificis amb factors $p < 0$, que són els que es troben darrere la càmera, els col·locarem a una llista separada posant p en valor absolut. Ordenarem cada llista de puntuacions de menor a major i començarem agafant els de la llista dels que estan per davant la càmera en ordre. Quan omplim un nivell de detall, passarem al següent i així successivament. Quan acabem amb la llista dels de davant, col·locarem els de la llista d'edificis que queden per darrere la càmera.

Per no carregar amb massa càlculs quan ens estem movent contínuament, no recalcularem el nivell de detall dels edificis cada vegada que es mogui la càmera. Concretament, hem fixat que es recalculi quan la càmera s'hagi mogut una distància igual o superior al radi mitjà dels edificis de l'escena.

Aquesta versió dels mapes de fotons requereix fer petits canvis al pas de fotons i al pas d'il·luminació. Concretament, voldrem saber de cada edifici quin és el seu nivell de detall per poder escriure i llegir del LOD corresponent del mapa de fotons. Per a més comoditat als accessos, definirem cada mapa de fotons com una textura 3D i cada capa sera un nivell de detall.

Les dues imatges següents mostren els fotons colorejats segons el nivell de detall utilitzat per a la façana de l'edifici. De major resolució a menor, els colors són: vermell, verd, blau i cian. Els fotons blancs són els del terra, que és uniforme. Fixem-nos també que els fotons d'un nivell de detall són 4 vegades majors que els del nivell anterior amb més resolució. La idea és que la distància a la càmara compensarà aquest efecte.

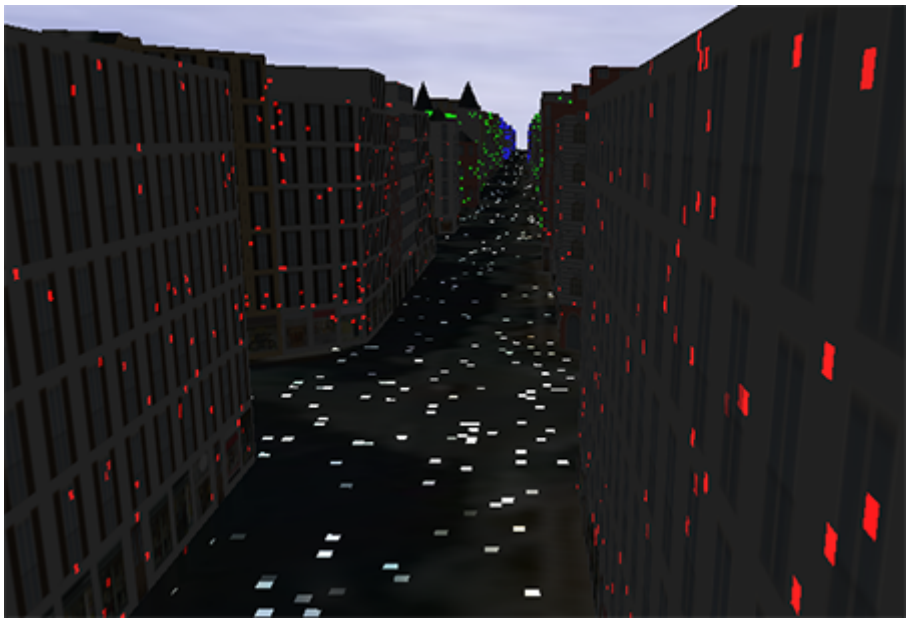

Figura 11.3: Un dels millors casos per a la variant adaptativa són les visualitzacions a nivell de carrer, ja que els edificis amb baixa resolució de fotons (colorejats en blau) queden bastant més allunyats que els que fan servir una bona resolució (en vermell).

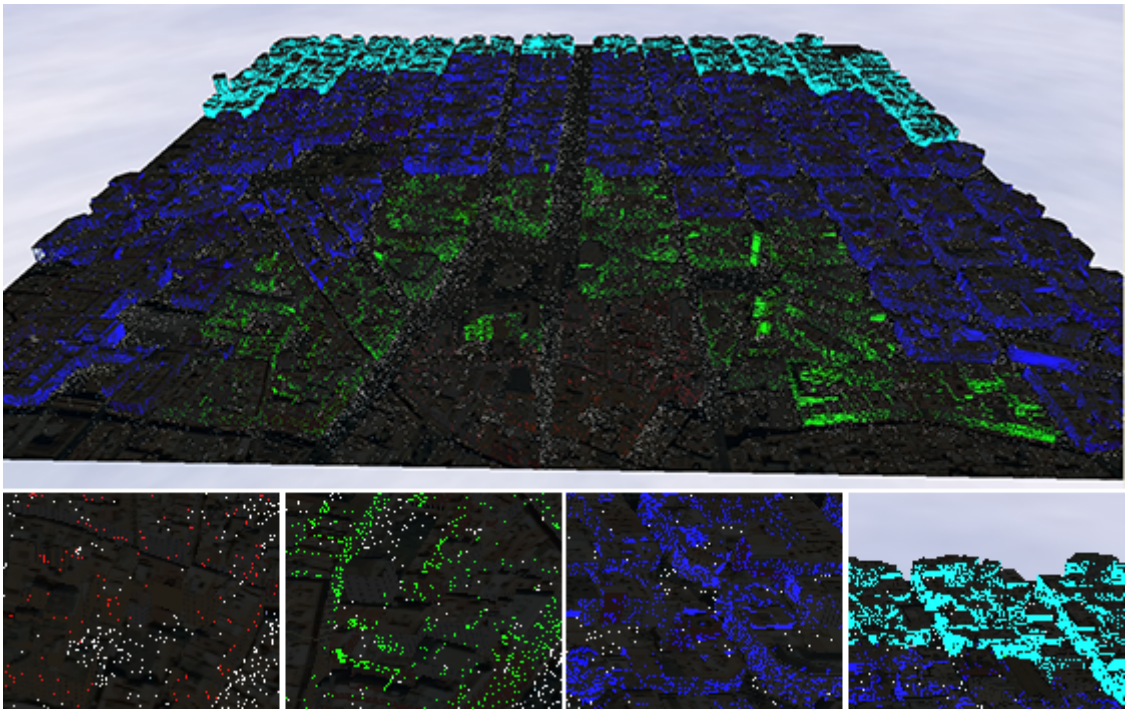

Figura 11.4: Classificació dels edificis en nivells de detall segons la distància i orientació respecte la càmera.

Capítol 12

Anàlisi i disseny

Aquest capítol descriu les diferents fases de l'enginyeria del software que s'han vist involucrades en el desenvolupament de l'aplicació. Partint d'un anàlisi dels requisits que havia de tenir el visualitzador, s'ha fet una especificació dels casos d'ús i un posterior disseny del projecte adaptat a les tecnologies utilitzades.

12.1 Anàlisi de requisits

Els requisits són el conjunt de funcionalitats que una aplicació ha de complir. Normalment aquests requisits s'haurien de definir conjuntament amb el client de l'aplicació, però al nostre cas l'aplicació té com a finalitat visualitzar els resultats dels algorismes que implementem.

Els **requisits funcionals** són els que indiquen què ha de fer l'aplicació. Hem definit els següents:

- Ser capaç de realitzar el preprocés del model de Barcelona per tal de generar les entrades necessàries per al nostre visualitzador (model i textures).
- Carregar el model i les textures resultants del preprocés. S'haurà de poder canviar el model dinàmicament sense reiniciar el programa.
- Visualitzar la ciutat carregada.
- Mostrar els resultats dels Photon Mapping original i del nostre algorisme.
- Navegar de manera interactiva pel model.
- Modificar en temps real les característiques de la il·luminació.
- Seleccionar l'algorisme a fer servir i modificar els paràmetres pertinents sense haver de reiniciar l'aplicació.
- Mostrar informació sobre el rendiment del programa i les característiques del model.

Els **requisits no funcionals**, en canvi, descriuen aspectes sobre com ha de ser programa i no sobre què ha de fer. S'han definit els següents:

- El programa ha de ser eficient: no fer càlculs redundants ni utilitzar més memòria de la necessària.

- També hauria de ser extensible, permetent incorporar noves variants d'algorismes a mesura que es vagin desenvolupant.
- S'ha de contemplar en la mesura que sigui possible l'escalabilitat quan les capacitats del hardware augmentin.
- Els resultats obtinguts pels algorismes d'il·luminació han de ser realistes.
- La visualització s'ha de poder fer, com a mínim, en temps interactiu (més de 5 imatges per segon).
- La interfície ha de ser intuïtiva i usable.
- S'ha de fer una documentació sobre el funcionament bàsic del programa.
- Hauria de poder funcionar en diversos sistemes operatius.

12.2 Especificació

De l'especificació n'obtenim per una banda els **casos d'ús** que descriuen aquelles tasques que l'usuari pot realitzar amb l'aplicació. A grans trets, els podríem agrupar de la manera següent:

- Interacció amb l'escena: fent servir teclat i ratolí l'usuari podrà navegar, fer *pan* i *zoom*, moure la càmera i canviar l'orientació.
- Carregar model: permetrà seleccionar de disc el fitxer amb el model del tipus concret suportat per l'aplicació, ja preprocessat.
- Canviar paràmetres de la llum: orientació, intensitat, zona a il·luminar, etc.
- Canviar paràmetres de l'algorisme: seleccionar entre el Photon Mapping original i el nostre algorisme, seleccionar el tipus de mapa de fotons per al nostre algorisme, canviar paràmetres de visualització.

D'altra banda, de l'especificació obtenim el diagrama de classes del sistema. Aquest diagrama, però, ja s'ha fet alhora que s'incorporaven els elements característics de l'etapa de disseny i ho comentarem tot junt a continuació.

12.3 Disseny

El projecte consta d'un total de tres mòduls: l'aplicació d'OptiX, l'aplicació d'OpenGL i una llibreria amb tot un conjunt de funcions i classes útils per a aplicacions gràfiques que utilitzen totes dues aplicacions, enllaçant-la dinàmicament fent servir la seva DLL.

La llibreria comuna, anomenada **Osu3D**, conté classes que implementen tipus de dades i funcions molt habituals per a aplicacions de visualització. Disposa de tipus de dades bàsics com vectors, matrius, capses englobants i tests d'intersecció. A partir d'aquests es formen les típiques d'estructuració d'una escena: vèrtexs, cares, objectes, escena, materials, càmera, etc. També conté classes per abstraure les crides que es realitzen a OpenGL, de manera que en un futur es podrien actualitzar a versions més recents sense afectar gaire

a la resta del disseny d'aplicacions. Finalment, hi ha els carregadors de models per a formats habituals com OBJ o PLY, i es van afegint a mesura que se'n necessiten més. És important destacar que part d'aquesta llibreria ja la tenia implementada abans de començar el projecte i part s'ha afegit o modificat durant la realització d'aquest.

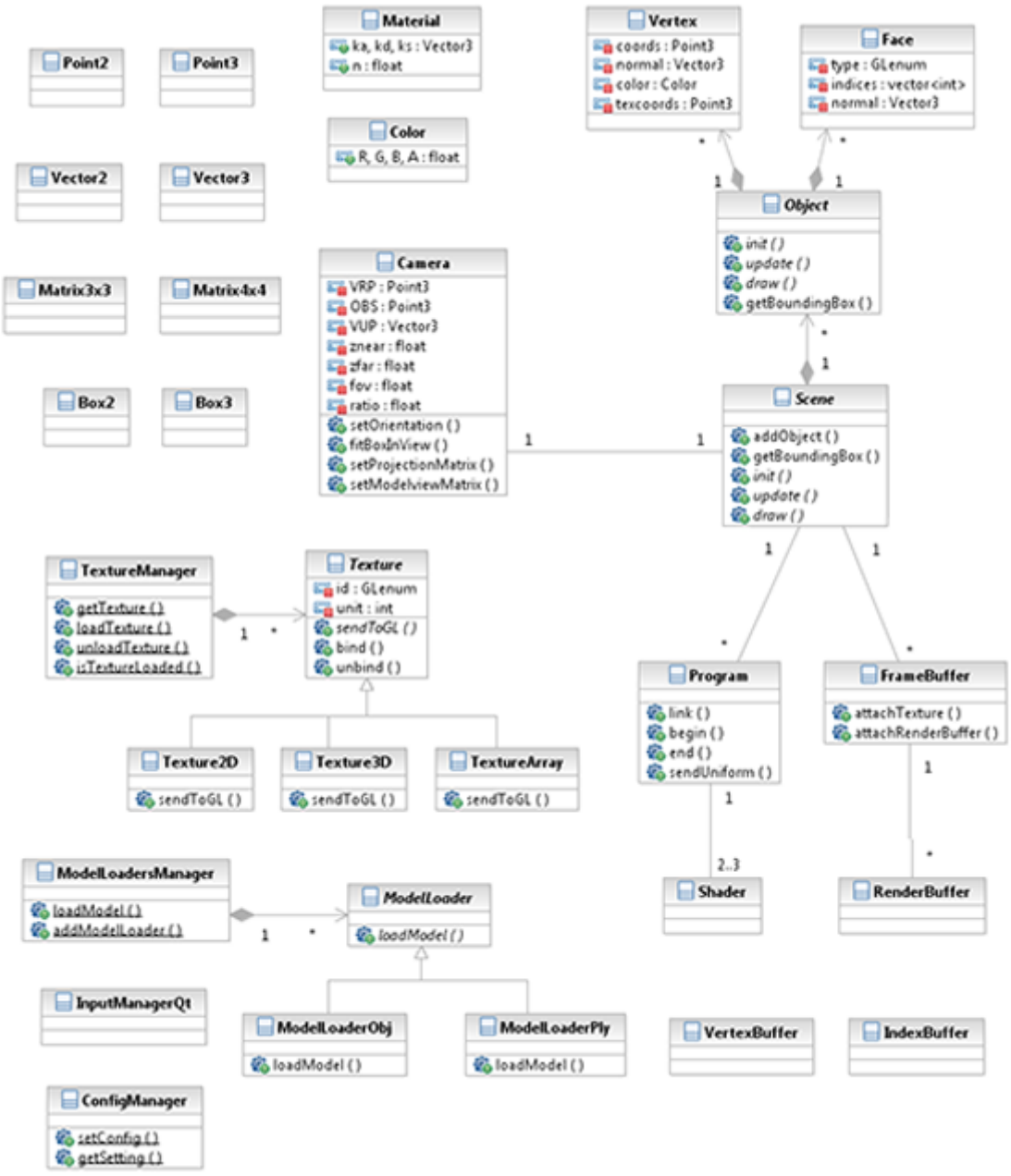

Figura 12.1: Diagrama de classes de la llibreria Osu3D, es mostren només els atributs i operacions més rellevants.

El **visualitzador OpenGL** és l'aplicació de visualització que es va desenvolupar per a la visualització i proves preliminars sobre el model, i que després s'ha fet servir per a implementar els algorismes del preprocés. Les classes més importants són *ModelLoaderBCN*, *ObjectBCN* i *SceneBCN*, que s'encarreguen de carregar el model de la ciutat i visualitzar-lo correctament. Per a les funcions relacionades amb el preprocés es disposa de les classes *SimplifierBCN* i *TextureCreatorBCN*.

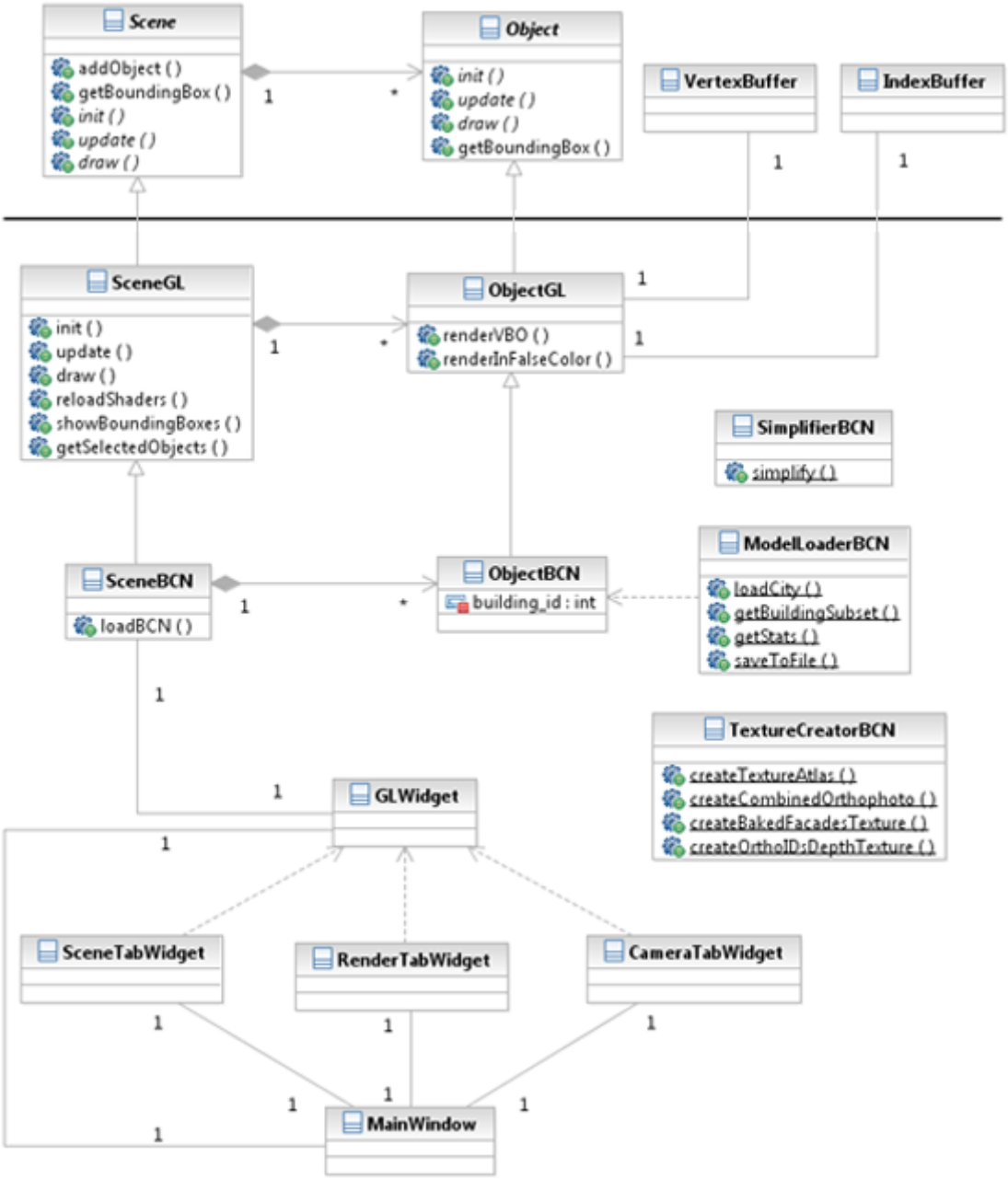

Figura 12.2: Diagrama de classes del visualitzador d'OpenGL, on es mostren només els atributs i operacions més rellevants. La part superior que es mostra separada són classes de la llibreria Osu3D.

Finalment, l'aplicació principal és el **visualitzador d'OptiX**. De manera anàloga a l'anterior, disposa de les classes *ObjectBCN* i *SceneOptiXBCN*, però ara ja no hereten d'una versió genèrica d'objectes i escenes d'OpenGL sinó que ho fan d'una versió genèrica d'aquests per a OptiX. Entre les classes més importants destaca també el *PhotonMapper*, que implementa l'algorisme original amb el *kd-tree*, i el *PhotonMapperBCN* que hereta de l'anterior i reimplementa l'estructura de dades fent servir els nostres mapes de fotons. La classe *Benchmarker* serveix de suport a la realització de tests automàtics. Com a observació, les textures també es veuen ampliades respecte la versió comuna d'OpenGL, ja que cal fer tractaments addicionals a OptiX per a carregar-les correctament.

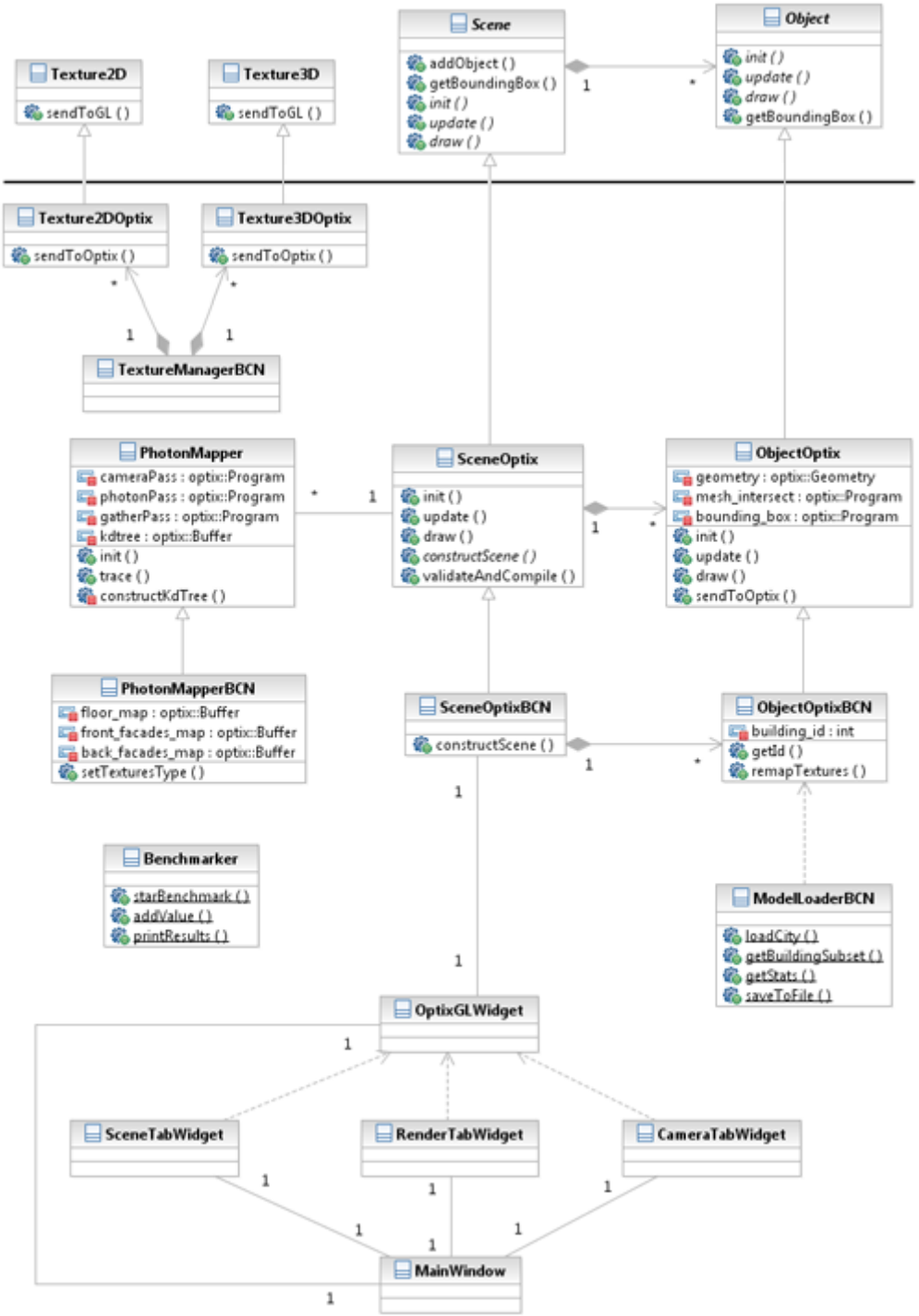

Figura 12.3: Diagrama de classes del visualitzador d'OptiX, on es mostren només els atributs i operacions més rellevants. La part superior que es mostra separada són classes de la llibreria Osu3D.

12.4 Tecnologies i eines utilitzades

La programació de l'aplicació s'ha realitzat en **C++**. La interfície fa servir **Qt 4.7**. Els visualitzadors fan servir **OpenGL 2.1**, en un cas íntegrament i a l'altre només per a mostrar el resultat del Photon Mapping. El visualitzador d'OpenGL fa servir *shaders* escrits en **GLSL 2.0**. Per al visualitzador d'OptiX s'ha fet servir la versió d'**OptiX 2.1**. Els programes associats a OptiX s'han programat fent servir **CUDA en C**.

El projecte s'ha programat i provat a *Windows XP 64bits* fent servir com a entorn de desenvolupament el *Microsoft Visual Studio 2005*. Per al disseny d'interfícies s'ha utilitzat el *Qt Designer*. Com cap de les tecnologies utilitzades pel projecte és depenent de la plataforma, ja que tant OpenGL, com CUDA i OptiX estan disponibles a Windows, Linux i Mac, es podria fer servir altres entorns de desenvolupament com per exemple *Qt Creator* i fer versions per a d'altres sistemes operatius.

Capítol 13

Resultats

A continuació, es mostren i discuteixen els resultats obtinguts d'acord amb els objectius inicials plantejats. Es veuran comparacions visuals i de rendiment amb la versió original del Photon Mapping progressiu i comentarem les limitacions que té el nostre algorisme i com creiem que podríem solucionar-les.

13.1 Visualització

Un dels objectius principals del projecte consistia en visualitzar models de ciutats fent servir un algorisme d'il·luminació global, i observar fenòmens associats a aquests algorismes com els descrits a la secció 3.2.2. Com es pot veure a les imatges següents, el nostre algorisme permet simular *color bleeding*, ombres, reflexions i il·luminació indirecta. Degut a la naturalesa del model, no s'observen càustiques, però se'n poden formar sense haver de modificar l'algorisme. Amb petits canvis al codi dels passos de càmera i de fotons, es podrien afegir superfícies transparents i observar refraccions.

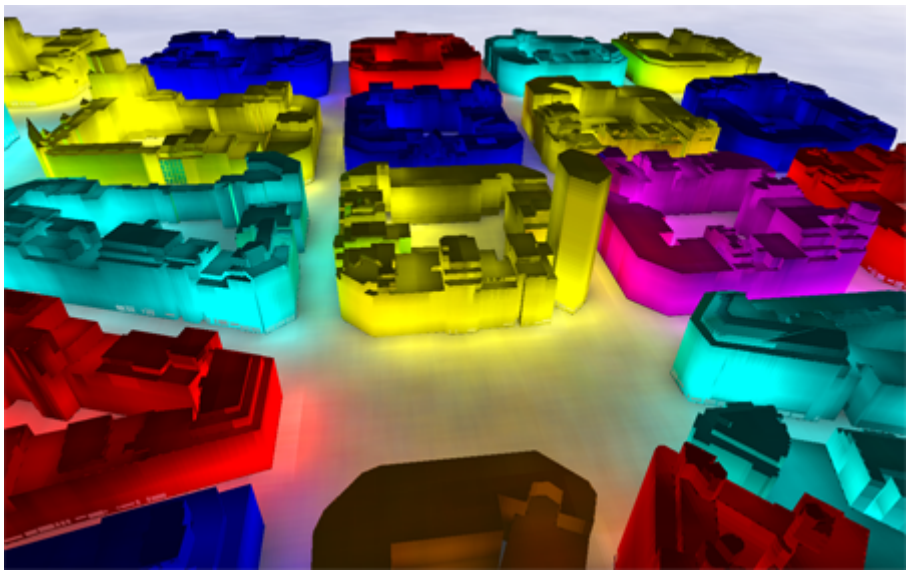

Figura 13.1: Imatge que mostra el fenomen del *color bleeding* al model. S'ha activat només la il·luminació indirecta amb la visualització en fals color dels edificis per a veure'l millor. A part de la propagació del color de façanes al terra, s'observa també com ho fa entre façanes d'edificis propers.

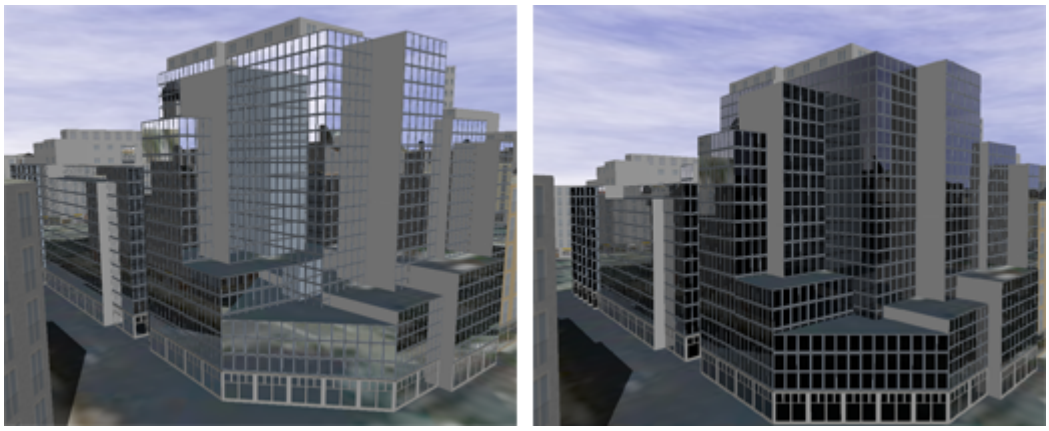

Figura 13.2: Dues visualitzacions diferents de les reflexions. A l'esquerra, les reflexions són sempre especulars pures i completament reflectants. A la dreta, s'aplica l'aproximació de Schlick pel factor de Fresnel, i la part no reflectida es pinta negra per a visualitzar millor el canvi de reflectància.

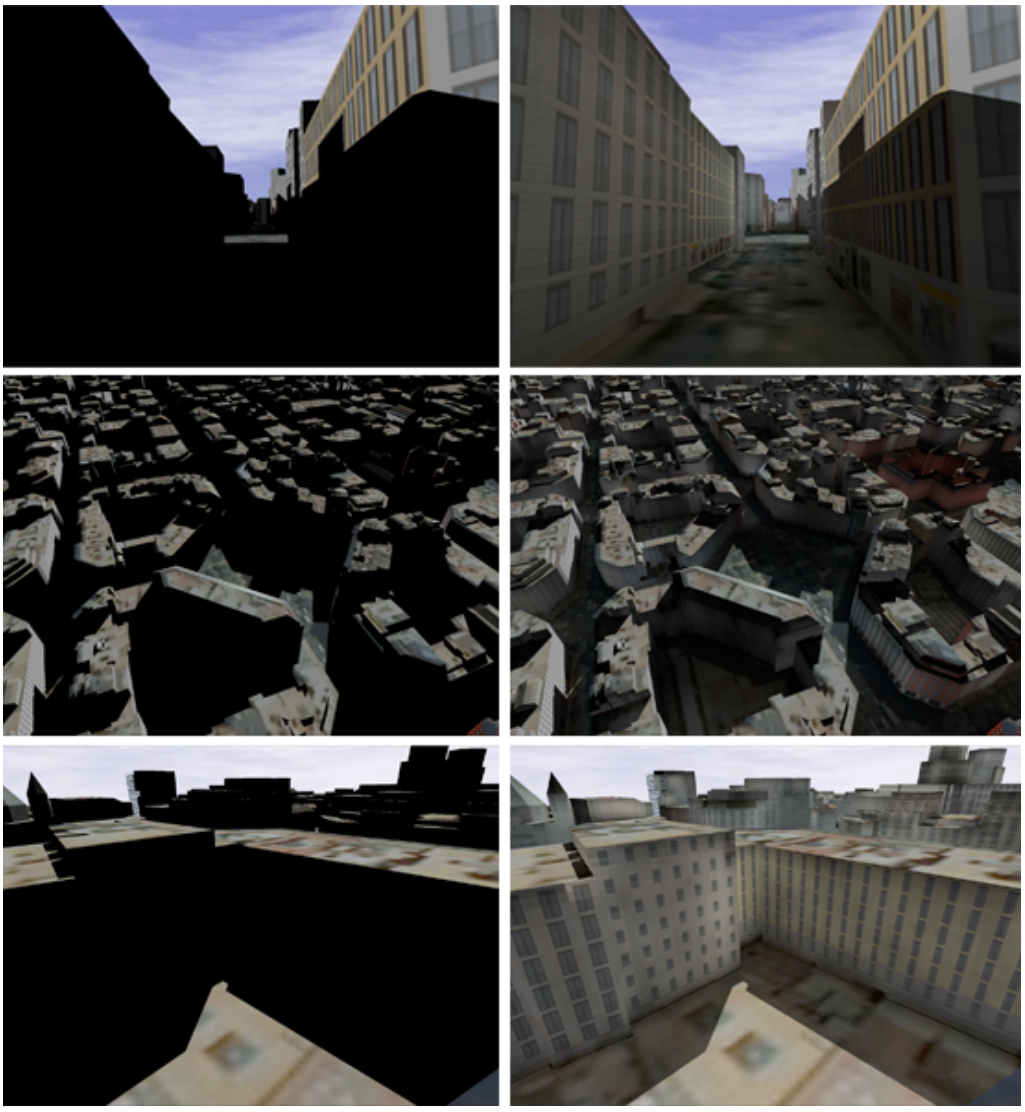

Figura 13.3: Diferents comparacions entre la il·luminació amb únicament llum directa (esquerra) i afegint la llum indirecta (dreta).

13.2 Comparació visual

Acabem de veure que el nostre algorisme és capaç de simular els diferents efectes que es produeixen amb la il·luminació global. El següent pas és comparar la qualitat visual amb la que obté l'algorisme original fent servir el *kd-tree*.

La figura 13.4 mostra una imatge obtinguda amb el nostre algorisme fent servir els tres tipus de mapes de fotons i amb l'algorisme original. Com es pot veure, gairebé no s'aprecien diferències.

Per a poder apreciar millor el canvi, podem visualitzar només aquella part que realment canvia entre els algorismes: la il·luminació indirecta. Per tant, desactivant la il·luminació directa i la texturació del terra podem fer més visible els càlculs de la il·luminació indirecta, com es mostra a la figura 13.5. Podem veure que la llum indirecta presenta imperfeccions, o errors, a totes les imatges, però la seva manifestació és bastant diferent entre els dos algorismes.

Amb l'algorisme original, el resultat del càlcul de llum indirecta sembla una textura com si fossin núvols. És el resultat de la superposició dels diversos cercles de radi r al voltant de cada fotó, ja que un fotó afectarà a tots els punts de les superfícies que estiguin a distància r o menor. Amb el nostre algorisme, en canvi, l'àrea d'influència de cada fotó és un rectangle alineat amb els eixos. A més, la posició de cada fotó ve discretitzada d'acord amb la resolució de la textura del mapa de fotons. Per tant, al superposar diversos rectangles amb posicions discretitzades podem observar millor les separacions entre l'àrea afectada per cada fotó. Tot i així, si no estem molt a prop aquestes separacions no s'aprecien.

La figura 13.6 mostra una altra escena comparada. A més, es mostra la millora obtinguda amb l'algorisme adaptatiu quan centrem l'atenció sobre un edifici concret.

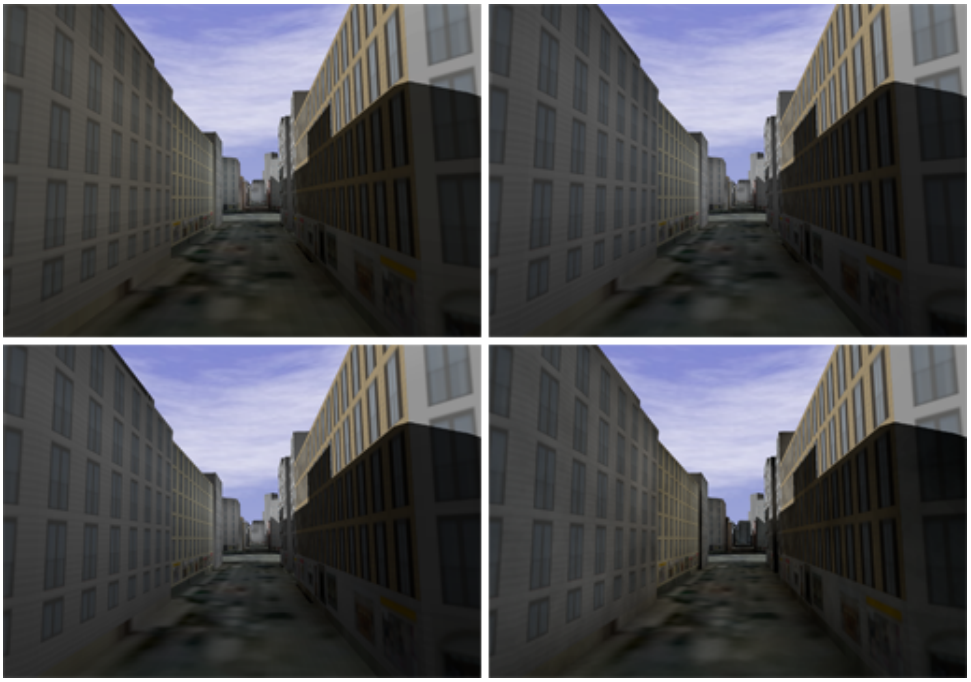

Figura 13.4: Comparació entre l'algorisme original i els nostres. A dalt a l'esquerra es mostra l'algorisme amb el mapa de fotons en color, a la dreta amb el de luminància, a baix a l'esquerra amb l'adaptatiu i a la dreta l'original amb el *kd-tree*.

Figura 13.5: Comparació de la component indirecta entre l'algorisme original i els nostres. A dalt a l'esquerra es mostra l'algorisme amb el mapa de fotons en color, a la dreta amb el de luminància, a baix a l'esquerra amb l'adaptatiu i a la dreta l'original amb el *kd-tree*.

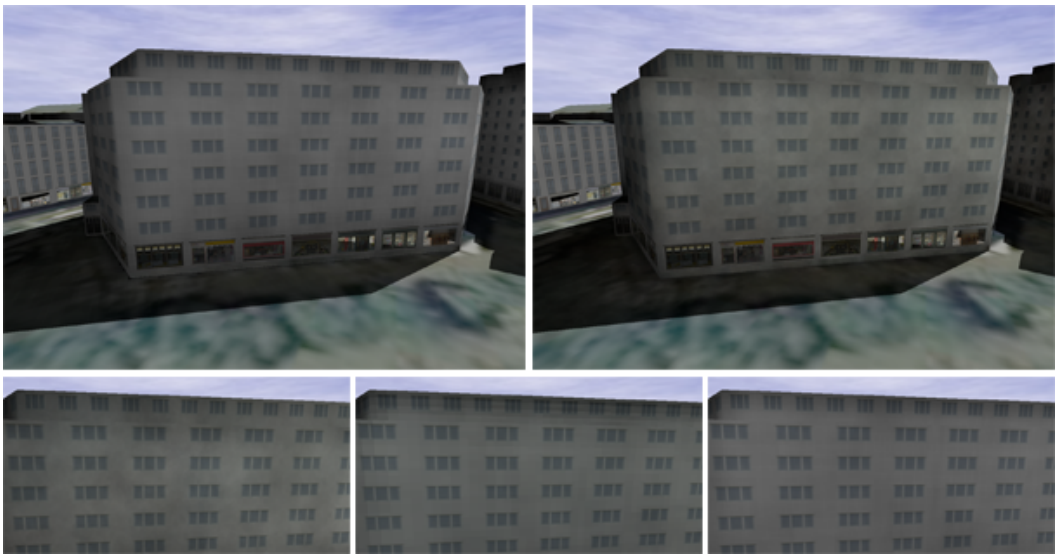

Figura 13.6: A dalt veiem dues imatges d'una façana generades amb la versió de mapes de fotons adaptatius i amb el *kd-tree*, respectivament. A sota, es mostra una ampliació de la component indirecta, per al *kd-tree*, mapa de fotons en color i mapa de fotons adaptatiu, respectivament.

Per tal de poder observar millor el principal origen de les diferències visuals entre els dos algorismes, la figura 13.7 mostra imatges dels edificis renderitzats en color fals i en les que s'està mostrant només la llum indirecta. Es pot veure com, efectivament, amb el nostre algorisme l'àrea d'influència d'un fotó és un rectangle o quadrat, mentre que amb l'original és un cercle. No obstant, tot i que es notin més les discretitzacions de les posicions dels fotons que introdueix la resolució de la textura, necessitem radis menors per a obtenir uns resultats que estiguin aproximadament igual de suavitzats que els de l'original. Veurem tot seguit, a la següent secció, que això és bo pel nostre algorisme, ja que el rendiment dependrà en bona part d'aquest radi de cerca.

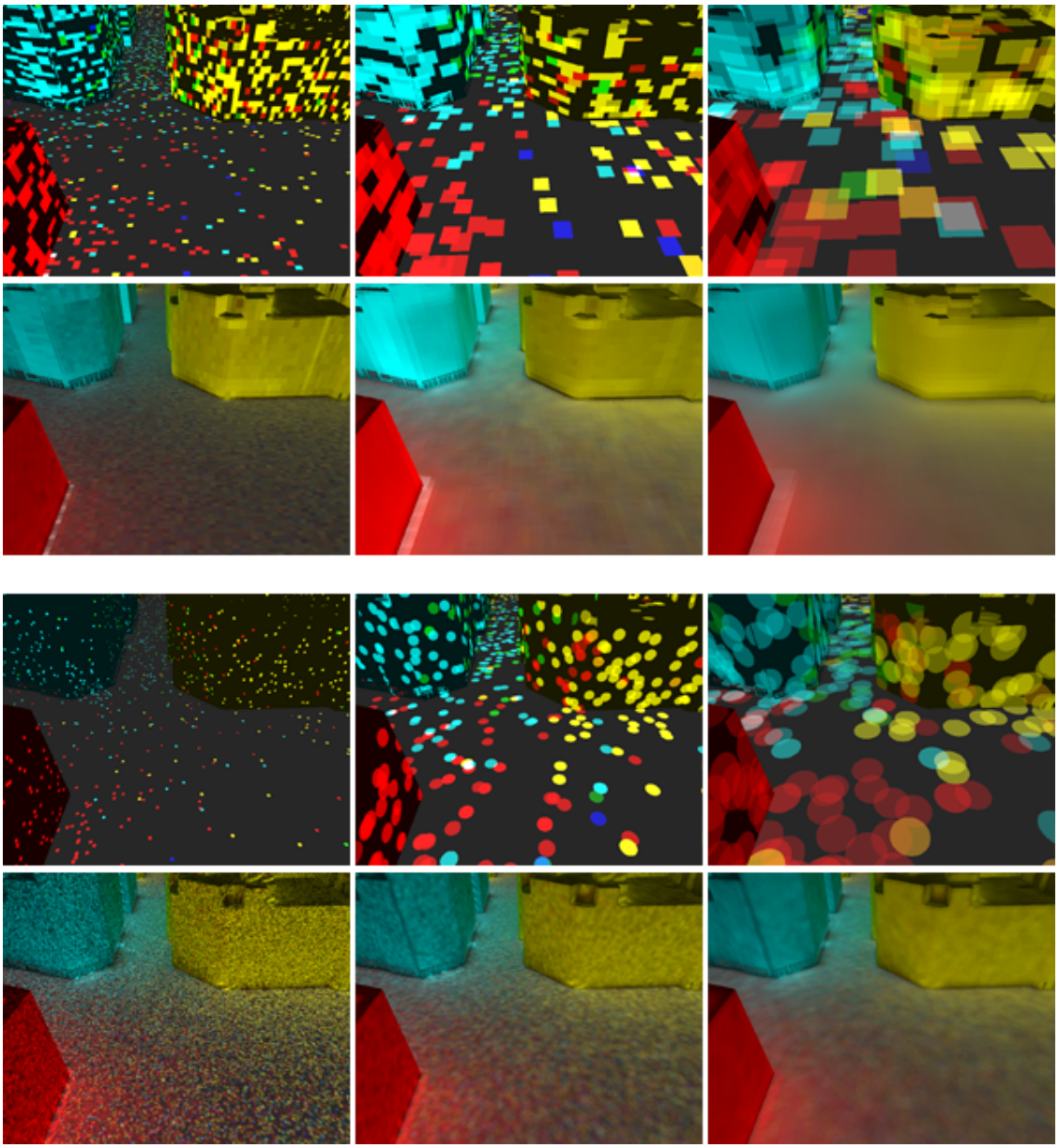

Figura 13.7: Comparació de la visualització dels radis de cerca de fotons. Les dues primeres files corresponen al nostre algorisme amb el mapa de fotons en color. Les dues últimes són les de l'algorisme original amb el *kd-tree*. Per a cada grup, es mostra el resultat d'una sola iteració i el de la situació gairebé estable, quan ja no s'observen gaire canvis amb el temps. D'esquerra a dreta, cada columna correspon a un radi $r = 0.3$, $r = 1.5$ i $r = 3.5$.

13.3 Comparació de rendiment

Per a veure els resultats de rendiment, s'ha fet servir un model més petit amb part del centre de Barcelona que conté 44 edificis, 18 773 triangles i 78 750 vèrtexs. S'han triat dos punts de vista prou diferents: l'escena que anomenarem A correspon a una vista aèria, mentre que l'escena B és una vista a peu de carrer. La figura 13.8 mostra aquestes dues vistes. Les proves han servit per a veure el rendiment dels quatre algorismes: l'original amb el *kd-tree* i les tres variants del mapa de fotons (color, luminància i adaptatiu).

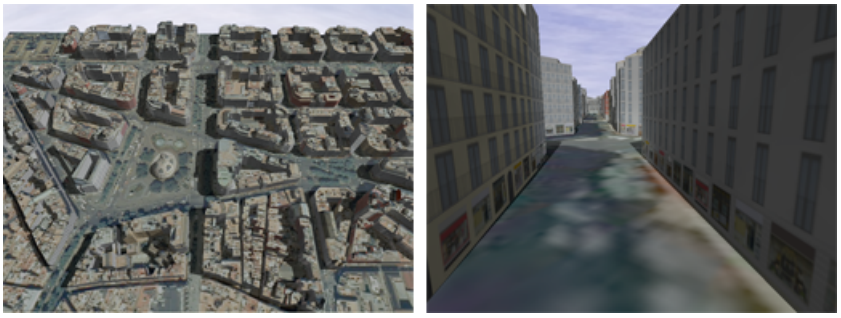

Figura 13.8: Les dues escenes utilitzades per a fer els *benchmarks* dels resultats.

El pas de fotons sempre llençarà 65 536 fotons, i permetem únicament dues reflexions difoses, però tantes d'especulars com es trobi durant el seu recorregut. Donat que el tipus de complexitat que introdueix llençar-ne més és la mateixa a tots quatre algorismes, no hem analitzat el comportament quan variem aquest nombre. Pel mateix motiu, el model sempre s'ha fet servir el mateix, ja que fer-ne servir de diferents modificaria només el temps de recorregut dels rajos per les estructures d'acceleració als passos de càmera i de fotons, i d'això n'és responsable completament OptiX.

Les proves s'han realitzat en un ordinador amb les característiques de la taula següent:

Processador	Intel Core 2 Duo E8500 (2 x 3.16 GHz)
Memòria RAM	8 GB
GPU	NVIDIA GeForce GTX 280
Memòria gràfica	1024 MB GDDR3
Sistema operatiu	Windows XP 64 bits

Taula 13.1: Característiques de l'ordinador on s'han fet els *benchmarks*.

13.3.1 Temps de cada pas

El primer que volem analitzar és el temps que triga a completar-se cada pas, i comparar-ho amb l'algorisme original. S'han fixat unes condicions comunes per a tots quatre algorismes: una resolució de 1024×768 píxels i un radi de 1 unitat. Es fan 50 iteracions, on la càmera es considera que ha canviat a cadascuna per tal de recalcular el pas de càmera. S'inclou el temps d'esborrat dels nostres mapes de fotons i el de construcció del *kd-tree* en l'algorisme original. Els resultats són els que mostren les taules 13.2 i 13.3.

Com es pot veure, el pas de càmera no varia entre els algorismes ja que és el mateix, però sí que es veu afectat pel tipus d'escena. És possible que si els rajos intersequen sovint amb les mateixes superfícies, com és el cas de l'escena B, la memòria *cache* de la targeta gràfica ajudi a millorar el rendiment dels tests. També pot ser que el recorregut de l'estructura d'acceleració sigui molt més semblant, i això redueix la divergència dels *threads* de CUDA i millora el rendiment.

Pel que fa al pas de fotons, els resultats corroboren la intuïció que desar més informació per cada fotó augmenta el temps del pas. Per això, els millors rendiments els tenim als mapes de fotons adaptatius i de luminància, que només guarden un *float* (4 bytes) per cada fotó a la textura, seguits dels mapes de fotons en color, que guarden 4 *floats* (16 bytes) i, finalment, el *kd-tree* que fa servir una estructura que ocupa un total de 16 *floats* (64 bytes).

De manera similar, el pas d'il·luminació també es veu millorat, tot i que amb els nostres algorismes sembla ser més dependent de l'escena. Mentre que amb l'algorisme original el recorregut del *kd-tree* sempre començarà pel mateix node, la nostra cerca de fotons comença directament al píxel corresponent a la projecció del punt. Per tant, com es veu si comparem l'escena A amb la B, la localitat espacial ajuda a millorar el rendiment, ja que entre dues posicions de la façana molt properes accediran probablement a la mateixa posició del mapa de fotons. També s'observa com el temps de construir el *kd-tree* és unes 3 vegades major al cost d'esborrar els mapes de fotons a cada pas.

	Càmera	**Fotons**	**Il·luminació**	***kd-tree***	**Esborrat**	**Total**
color	57.14	32.22	13.40		4.37	107.13
luminància	52.33	28.65	16.92		4.34	102.24
adaptatiu	51.32	28.92	17.00		4.34	101.58
kd-tree	52.89	35.65	17.89	12.84		119.27

Taula 13.2: Temps de cada pas a l'escena A, en mil·lisegons.

	Càmera	**Fotons**	**Il·luminació**	***kd-tree***	**Esborrat**	**Total**
color	36.43	31.09	14.45		4.38	86.35
luminància	39.29	30.86	14.54		4.40	89.09
adaptatiu	37.72	30.79	14.90		4.39	87.80
kd-tree	38.28	34.73	18.31	13.05		104.37

Taula 13.3: Temps de cada pas a l'escena B, en mil·lisegons.

13.3.2 Radi de cerca

La següent prova que hem fet és observar com el radi de cerca dels fotons propers afecta al rendiment. Per a fer aquesta prova, s'ha fixat una resolució de 1024×768 píxels. S'han provat 15 radis diferents, incrementant de 0.5 en 0.5 el valor del radi des de 0 a 7 unitats. Per a cada radi, s'han fet 50 iteracions del pas de fotons i d'il·luminació, calculant els mil·lisegons a cadascuna i fent posteriorment la mitjana. El factor α de reducció del radi s'ha fixat a 1, de manera que sempre es manté el mateix radi al llarg de les iteracions i podem una bona aproximació del temps que triga. Les gràfiques de la figura 13.9 mostren els resultats obtinguts.

La primera observació que se'n pot fer és que l'algorisme original no es veu afectat gens pel radi de cerca. En canvi, als nostres algorismes, quan aquest augmenta cal fer més accessos a les textures de fotons. Lògicament, com als mapes de luminància i adaptatius l'àrea d'un píxel és menor, el rendiment baixa més que amb els de color. Tot i això, al principi el rendiment és major, probablement a causa de que estem llegint un sol *float* de les textures en comptes de quatre i la manera en com OptiX organitza internament el codi de CUDA en *threads* per a fer els accessos a posicions properes. Quan augmentem el radi, ja fa falta llegir diversos píxels de la textura i es nota abans aquest increment d'accessos,

que és quadràtic respecte el radi. S'observa també molt clarament a la gràfica dels mapes de fotons en color com la discretització de l'àrea dels fotons fa que els salts al rendiment siguin discrets, concretament produint-se quan cal agafar més píxels pel nou radi.

Si comparem ara les dues escenes entre elles s'observa que els nostres algorismes es veuen afavorits per escenaris com el B, igual que hem vist a la prova anterior, mentre que l'algorisme original es manté intacte. És també bastant probable que el motiu d'aquest canvi siguin els accessos a les textures, ja que ara hi ha poques façanes i la majoria de píxels accediran als mateixos píxels o molt propers del mapa de fotons. Per tant, tenim molta coherència espacial i la memòria *cache* de la gràfica pot ajudar a incrementar el rendiment, llegint blocs de píxels per servir diversos *threads* alhora.

Veiem que totes dues gràfiques hi ha un punt a partir del qual el radi creix massa i el rendiment passa a ser pitjor que amb l'algorisme original. No obstant, aquest punt correspon a un radi que ens proporciona en la majoria de casos resultats d'il·luminació indirecta satisfactoris, com s'ha mostrat a la figura 13.7.

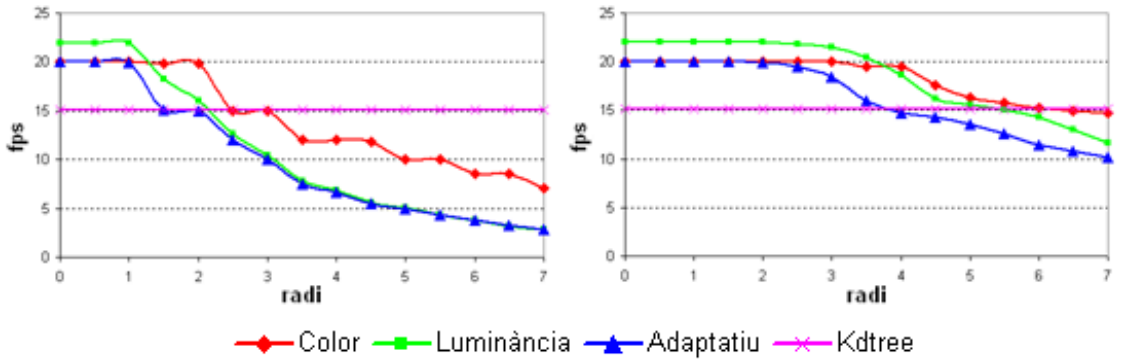

Figura 13.9: Gràfiques del rendiment, mesurat en *frames per segon*, per l'escena A (esquerra) i B (dreta) en funció del radi.

13.3.3 Resolució

Finalment, s'ha provat com afecta al rendiment la mida de la pantalla on renderitzem la imatge. Lògicament, aquesta només hauria d'afectar al rendiment del pas de càmera i al del pas d'il·luminació. Ens interessarà sobretot aquest últim, ja que ens permetrà veure com escalen els algorismes augmentant el nombre de píxels des d'on fer la cerca de fotons.

S'han provat diverses resolucions d'ús habitual i la màxima que podíem aconseguir maximitzant el programa: 800×600, 1024×768, 1280×720, 1280×900 i 1400×980. El radi de cerca s'ha fixat a 1. S'han fet 50 iteracions, executant sempre tots els passos (incloent càmera) i s'ha calculat la mitjana per a obtenir el resultat final. Els resultats obtinguts es mostren als gràfics de la figura 13.10.

Si ignorem el pas de càmera, podem veure com tots tres algorismes presenten una reducció lineal quan augmentem el nombre de píxels, a causa del major nombre d'execucions del pas d'il·luminació. Tots quatre algorismes presenten un comportament similar, com es veu a les gràfiques de la columna esquerra. Quan calculem el rendiment tenint en compte el temps d'execució del pas de càmera, el rendiment baixa bastant, com ja hem comentat abans, però manté bastant bé el decreixement lineal.

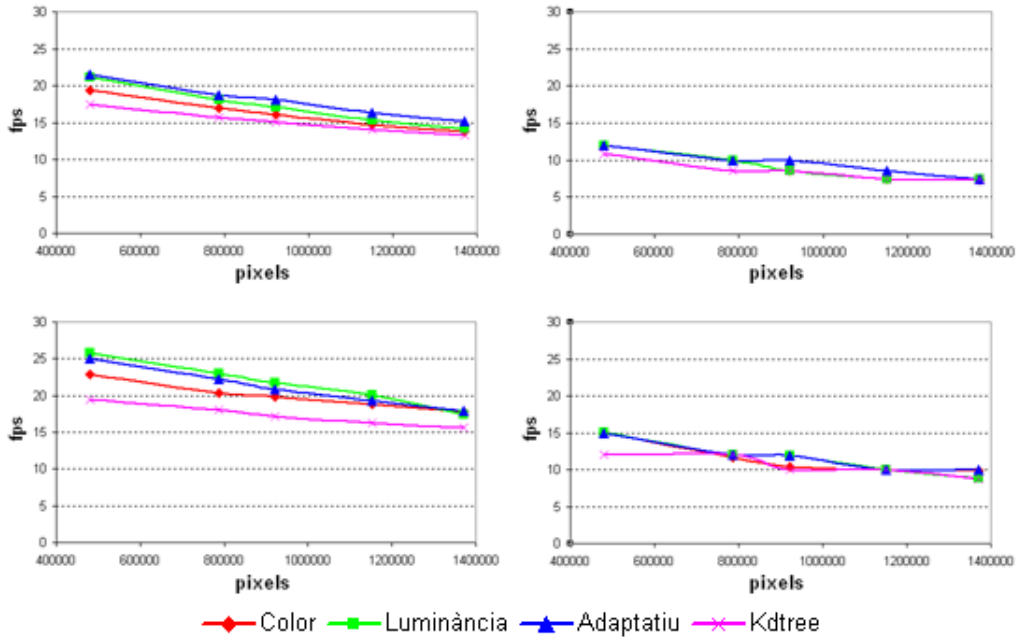

Figura 13.10: Gràfiques del rendiment, mesurat en *frames per segon*, per l'escena A (a dalt) i B (a baix) en funció del nombre total de píxels. A la columna esquerra no es té en compte el pas de càmera, mentre que la de la dreta sí.

13.4 Discussió

Per acabar, després de veure els resultats obtinguts discutirem les limitacions actuals del nostre algorisme i com creiem que podrien ser solucionades.

Com veiem a les taules del temps de cada pas de l'algorisme, el pas de càmera pot trigar entre uns 30 i 50 mil·lisegons. Quan triga 50 mil·lisegons, només aquest pas ja estaria limitant el rendiment a 20 fps. Si, a més a més, li afegim el temps dels altres passos, aquest rendiment baixa fins als 10 fps. En general, el pas de càmera es recalcula només quan la càmera es mou, canvia la seva orientació o es canvia alguna configuració del *render* que impliqui recalcular de nou tota la il·luminació. A més a més, si ens estem contínuament movent, només deixarem temps per a una iteració del pas d'il·luminació, de manera que es veuran gairebé els fotons individuals com a cada *frame* canvien les seves posicions. Una alternativa seria no fer el pas de fotons ni calcular la llum indirecta quan la càmera s'està movent, de manera que podríem seguir mantenint rendiments interactius per a navegar. Un cop es pari el moviment, començaria el pas de fotons i d'il·luminació amb llum indirecta, que combinats també ens donen uns 50 mil·lisegons.

A causa de la parametrització cilíndrica que hem escollit, hi ha dues limitacions imposades a la forma dels edificis. D'una banda, les façanes exteriors i interiors han de ser projectables per complet. És a dir, que una secció de façana en forma de U, on hi hagi per a una mateixa direcció angular dues parets a diferent distància del centre orientades cap a l'exterior (o interior) farà que només la més exterior (o interior) es vegi reflectida al mapa de fotons. Conseqüentment, tots els fotons que arribin a una de les dues parets es veuran també aplicats sobre l'altra. Afortunadament, degut a que això implicaria que les finestres d'una de les parets només veurien la paret de davant, és molt inusual.

L'altre problema relacionat amb la projecció cilíndrica són les parets que tenen una orientació gairebé paral·lela al radi que va dirigit al centre de la capsa englobant de l'edifici.

Com tenen molt poc increment d'angle, encara que sigui una paret llarga tots els punts es projecten sobre una mateixa posició del mapa de textures. Per tant, els fotons es veuen molt estirats horitzontalment. Tot i això, si estem visualitzant alhora llum directa i indirecta o el radi d'influència dels fotons és prou gran, no es nota l'efecte d'aquests fotons.

De manera similar, és prou obvi que la mida de les textures i el nombre d'edificis definiran el nombre de píxels assignats a cada edifici, afectant a la definició que obtindrem per a cada fotó. Quan tenim molts edificis o textures petites, els fotons es veuen massa grans. Com s'ha vist, les solucions adaptatives ens donen resultats prou bons. Es podria estudiar també aplicar algun filtrat quan s'agafen els fotons, de manera semblant a com fan els filtres de magnificació de textures. Una altra opció seria aplicar les idees dels algorismes de *Final Gathering* i considerar els nostres mapes de fotons com una solució de baixa resolució. D'aquesta manera podríem estalviar memòria i fer servir fotons més grossos a textures més petites, però el traçat de nous rajos des de cada superfície difosa també implicaria un temps addicional a cada *frame* que ens faria baixar el rendiment.

Com hem vist a la secció anterior, l'algorisme original no es veu afectat per la mida del radi dins el qual busquem fotons. En canvi, per al nostre algorisme, un radi major fa créixer quadràticament el nombre de píxels a llegir de la textura. S'hauria d'intentar mantenir el nombre de píxels llegits sempre per sota d'un llindar. Una idea seria fer un *splatting* dels fotons i afegir el seu flux a tots aquells píxels del voltant que sabem que estan a distància menor al radi. Com el nombre de fotons que es llencen a cada iteració és molt menor comparat amb el de píxels a il·luminar, estarem traslladant el cost quadràtic d'accedir a les textures a on es farà menys vegades i podríem obtenir temps menors. Altres idees per a mantenir el nombre d'accessos constant per cada píxel serien construir una *Summed Area Table* o una col·lecció de nivells de *Mip-mapping*, de manera que puguem accedir al nivell de detall desitjat en funció de la mida del radi.

Finalment, el motiu pel qual no hem pogut visualitzar el model sencer de Barcelona ha estat que la targeta gràfica no tenia prou memòria. Tot i tenir 1GB de memòria gràfica, OptiX necessita que totes les dades estiguin carregades abans de començar els traçats, i a més necessita memòria addicional per a les seves estructures d'acceleració i pel traçat. Es podrien intentar aplicar tècniques de visibilitat per a reduir el nombre d'edificis que hem d'enviar a OptiX sabent que no ens deixem cap de visible. No obstant, fins i tot si assumíssim que el cost de canviar la geometria a OptiX arribés a ser nul, a les visualitzacions aèries el problema persistiria, ja que tota la geometria s'ha de mostrar. Per a aquest cas, caldria aplicar probablement tècniques de simplificació.

Capítol 14

Planificació i anàlisi econòmic

Per a qualsevol projecte, sempre és important fer una valoració tant del temps com dels costos associats. Aquest capítol descriurà la planificació que s'ha seguit durant l'elaboració del projecte i farà una estimació del cost econòmic que suposaria realitzar-lo pagant els recursos.

14.1 Planificació

El projecte s'ha dividit en tres parts bastant diferenciades: la lectura i recerca de coneixements previs i estat de l'art, el desenvolupament del visualitzador i la redacció d'aquesta memòria.

Durant la primera part, que es va començar a principis de setembre, es va començar fent lectura de bibliografia bàsica sobre il·luminació global per adquirir els coneixement necessaris. Es va seguir fent investigació dels Raytracers interactius que havien aparegut durant els últims anys, fent especial èmfasi en analitzar detalladament OpenRL i OptiX. Quan es va decidir utilitzar com a base l'algorisme de Photon Mapping, es va aprofundir en la bibliografia relacionada amb aquest algorisme. Com el projecta ja s'anava encarant a fer servir OptiX a la nostra implementació, es va fer un estudi d'exemples i diverses proves per a familiaritzar-se tant amb CUDA com amb l'API d'OptiX.

El desenvolupament del visualitzador va començar paral·lel a les últimes fases de recerca. Mentre a les reunions setmanals s'anava perfilant com seria el nostre algorisme, s'anava fent ja el tractament i preprocés del model fent servir el visualitzador d'OpenGL. Un cop acabat l'estudi d'OptiX, es va implementar l'algorisme de Photon Mapping original per a visualitzar el model de Barcelona, alhora que s'adquiria més experiència de desenvolupament en CUDA i OptiX. El següent pas ja va ser la implementació del nostre algorisme, seguit de les extensions que es van proposar després. Aquest va requerir realitzar una mica més de preprocés addicional. Un cop es va tenir tot acabat, es van fer diverses proves de rendiment i es va anar optimitzant el codi fins a aconseguir els resultats que s'han presentat en aquesta memòria.

Finalment, ja s'havia previst des de mitjans del projecte de disposar d'un mes sencer per a poder escriure la memòria i tenir temps per a preparar la presentació de la defensa del projecte.

La figura 14.1 mostra el diagrama de Gantt de les diferents fases del projecte. S'ha calculat l'extensió aproximada de cadascuna en setmanes.

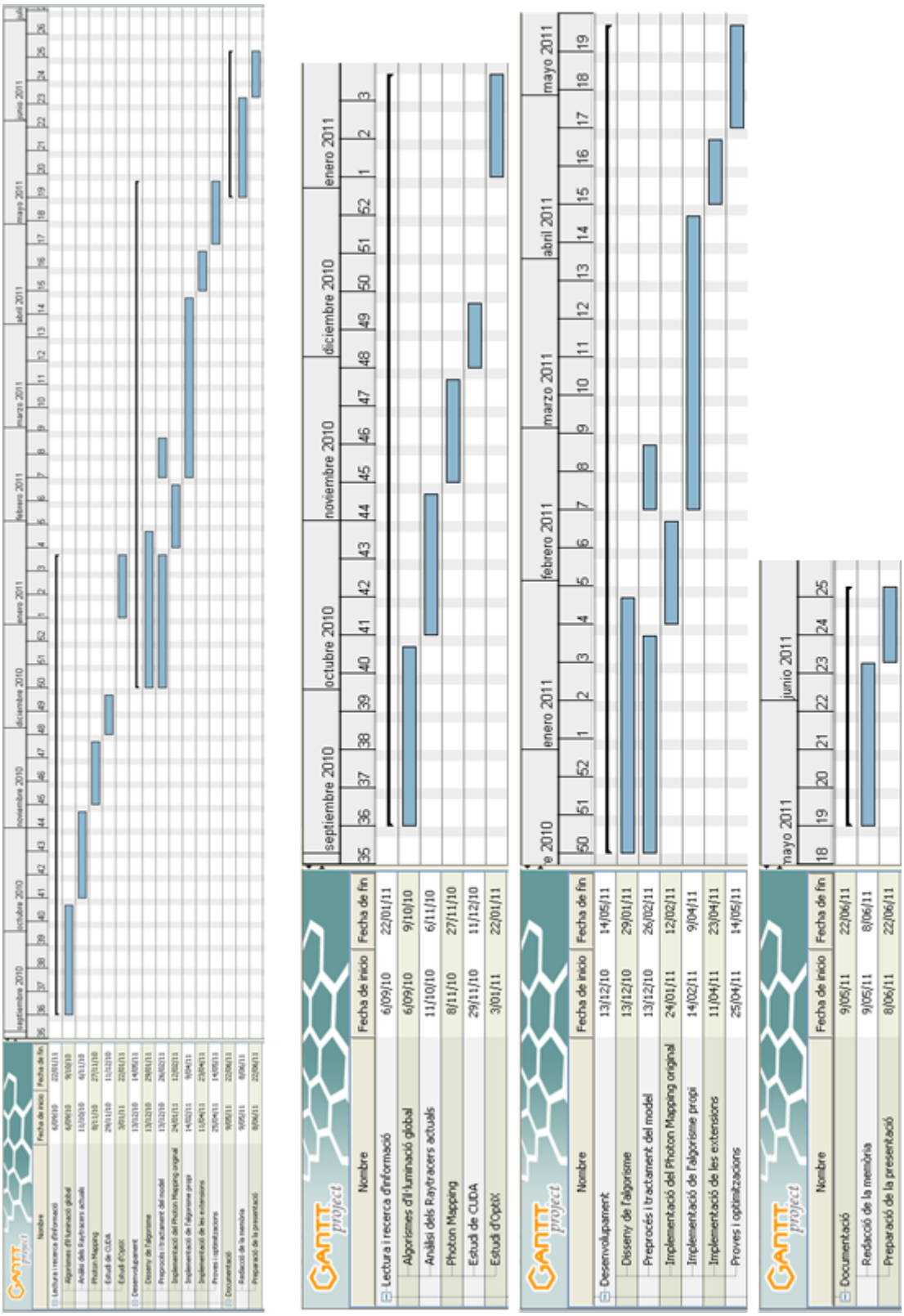

Figura 14.1: Diagrama de Gantt amb la planificació del projecte. Es mostra el diagrama complet i després tres ampliacions per a facilitar la lectura.

14.2 Anàlisi econòmic

La valoració dels costos econòmics la separarem en dues parts: els costos de recursos humans i els costos de maquinari.

Per a calcular els **costos de recursos humans** s'ha associat un perfil o rol de treballador a cadascuna de les tasques. Tenim dos tipus de perfil, cadascun amb el cost per hora següent:

Cost analista / dissenyador:	45 €/ h
Cost programador:	30 €/ h

S'ha fet una aproximació de les hores dedicades a cada tasca, i se'ls ha assignat el perfil que seria més habitual per a dur-la a terme. La taula següent mostra el desglossament del cost total dels recursos humans:

Tasca	**Perfil**	**Hores**	**Cost**
Lectura i recerca	analista/dissenyador	100	4500 €
Estudi de CUDA	analista/dissenyador	20	900 €
Estudi d'OptiX	analista/dissenyador	40	1800 €
Disseny de l'algorisme	analista/dissenyador	70	3150 €
Preprocés i tractament del model	programador	80	2400 €
Implementació Photon Mapping original	programador	60	1800 €
Implementació de l'algorisme propi	programador	200	6000 €
Implementació de les extensions	programador	20	600 €
Proves i optimitzacions	programador	30	900 €
Memòria	analista/dissenyador	80	3600 €
Total		700	25 650 €

Taula 14.1: Costos de recursos humans.

Pel que fa als **costos de maquinari**, només fa falta tenir en compte el cost de l'ordinador. L'element realment determinant pel resultat d'aquest projecte és la targeta gràfica, la qual hauria de ser NVIDIA per a assegurar el bon funcionament d'OptiX i CUDA. La millor targeta actual de gamma alta és la GeForce GTX 580, que podem trobar per uns 480€[1]. La resta de components no seran tant importants, ja que gairebé tot s'executa a la tarja gràfica i qualsevol mòdul de memòria que puguem trobar al mercat ens serà suficient per a permetre carregar el model. Tot i així, per aprofitar millor que la targeta és de gamma alta, afegint uns 2000€ més es pot completar perfectament un ordinador bo de gamma alta. Per tant, el cost de maquinari el podem estimar en uns 2500€.

S'ignora el cost de les eines de software utilitzades, ja que tot i que s'ha fet servir Windows i Visual Studio per a desenvolupar el projecte, es podria fer perfectament a d'altres sistemes operatius i amb entorns de desenvolupament lliures.

Per tant, el **cost total** del projecte podem dir que ha estat d'uns 28 150 €.

[1] http://www.pccomponentes.com

Capítol 15

Conclusions i treball futur

Els objectius principals del projecte, com es comentava a la introducció, eren aplicar un algorisme d'il·luminació global per a models d'entorns urbans i aconseguir rendiments interactius a la visualització. Les imatges que s'han mostrat al llarg del capítol de resultats són una prova que el nostre algorisme calcula correctament la il·luminació global per a una escena urbana. L'anàlisi del rendiment mostra que obtenim *framerates* superiors als de l'algorisme original, i ens alguns casos fins i tot els podem considerar que són en temps real. Per tant, podem afirmar satisfactòriament que no només s'ha aconseguit complir els objectius amb èxit, sinó que a més hem millorat una tècnica coneguda.

Seria interessant tenir la possibilitat de provar el visualitzador amb alguna targeta de gamma alta de la generació actual, la sèrie GTX 500, ja que els anàlisis que s'han fet d'aquestes targetes mostren un rendiment de les aplicacions en CUDA entre 2 i 4 vegades el de les targetes de la sèrie 200, com la que s'ha fet servir durant la realització d'aquest projecte.

Independentment de les capacitats del hardware gràfic, hem comentat les limitacions actuals del nostre mètode i les idees que creiem que ens podrien ajudar a solucionar-les. Molt probablement, els propers objectius seran reduir l'impacte del radi de cerca al rendiment i accelerar la convergència del càlcul de la llum indirecta, per exemple focalitzant més fotons sobre les façanes directament visibles, per a què en menys iteracions aquesta es mostri ja estable.

Un altre propòsit és aconseguir la visualització de models de mides majors, el que gairebé amb total seguretat implicarà aplicar tècniques de *culling* de la geometria o simplificacions. Per a fer-ho, caldrà veure com podem fer que el cost d'afegir, treure o modificar objectes de geometria a OptiX sigui el mínim possible.

Com a valoració personal, estic molt satisfet de la feina realitzada i, sobretot, dels resultats obtinguts. Per una banda, m'ha permès ampliar molt els meus coneixements sobre generació d'imatges realistes i sobre gràfics en temps real, dues àrees dels gràfics en les que tinc especial interès. Per l'altra, investigar i voler millorar un algorisme conegut ha estat des del principi un repte molt engrescador, i ha fet que l'esforç que implica l'elaboració d'un projecte final de carrera no fos en cap moment una càrrega, sinó més aviat una afició.

Per acabar, m'agradaria agrair a totes aquelles persones que d'una manera o altra han fet possible la realització i l'èxit d'aquest projecte. Primer de tot, als directors del projecte, Carlos Andújar i Gustavo Patow, per la seva supervisió del projecte, les seves idees i l'ajuda oferta. A la família, les amistats i, especialment, a la meva parella, per escoltar-

me pacientment i mostrar interès durant tot el projecte. Als membres de l'associació VGAFIB, per tot el que vaig aprendre quan encara estava al principi de la carrera, crear videojocs m'ha dut a especialitzar-me i a despertar la meva passió pels gràfics. I, finalment, als companys del grup MOVING, per la seva ajuda i experiència quan he tingut dubtes, estalviant-me així molt de temps i futurs problemes.

Part III

Annexos

Apèndix A

Pipeline d'OpenGL

OpenGL és una especificació estàndard que defineix una API multillenguatge i multiplataforma per a fer aplicacions gràfiques. Definim el *pipeline* com el conjunt d'etapes i operacions que es realitzen sobre les dades d'entrada. A continuació, s'explica com és aquest *pipeline* per a OpenGL 2.1 i superiors [37, 35]:

1. La geometria s'envia a la GPU. Normalment, les dades a enviar per cada vèrtex de la malla són: posició, normal, color i coordenades de textura. La millor manera d'enviar-les és fer servir un *Vertex Buffer Object*, ja que així es copiaran a la memòria de la GPU.

2. Per cadascun dels vèrtexs, s'executa el *Vertex Shader*. Aquest és el programa que s'encarregarà de transformar el vèrtex que rep a l'entrada per tal de passar les coordenades expressades en espai de món a coordenades en espai de *clipping*. També pot interessar expressar les normals en coordenades d'observador i transformar les coordenades de textura. Addicionalment, podem definir per cada vèrtex variables que voldrem que siguin interpolades entre els diferents vèrtexs d'un triangle, per exemple per a càlculs d'il·luminació.

3. La següent etapa és el *Primitive Assembly*. Els vèrtexs que surten del *vertex shader* s'agrupen formant les primitives especificades (triangles, quadrats, línies, ...).

4. Per a cadascuna de les primitives, s'executa opcionalment el *Geometry Shader*. L'entrada d'aquest serà la primitiva, per a la qual podrem recórrer els seus vèrtexs, i informació d'adjacència. La sortida pot ser zero o més primitives, de manera que podem crear i descartar geometria.

5. Tot seguit, i per a maximitzar l'eficiència de les etapes posteriors, es fa *clipping* (retallat) de les primitives, de manera que les que són completament fora de la regió visible per la imatge seran eliminades, i les que són parcialment dins seran retallades. També podem decidir activar el *culling*, que farà que els triangles que no estiguin mirant a la càmera siguin descartats (també es pot demanar que es descartin els que sí que miren en comptes dels que no).

6. A continuació, es fa la *rasterització*. Per cadascuna de les primitives que tenim en aquest punt, es calculen els píxels cobreix i es genera un *fragment* per cadascun d'ells. Els valors que s'hagin d'interpolar entre vèrtexs, com per exemple el color, la profunditat o les coordenades de textura seran també interpolats ara a cada fragment generat.

7. Per a cada fragment produït a la rasterització, s'executa el *Fragment Shader*. L'objectiu d'aquest és calcular el color final que haurà de pintar-se al *Framebuffer*, i opcionalment podem modificar la seva profunditat.

8. Finalment, es realitzen els tests i operacions de fragment. D'entre els diversos tests que podem activar, el més destacat és el *z-test*: si la profunditat del fragment és major que la del que conté el *Framebuffer* a la mateixa posició, el fragment es descarta ja que està per darrere i no és visible. Cal destacar que si el *fragment shader* no modifica el valor de la profunditat, aquest test s'executa abans per si ens podem evitar d'executar el *shader*. Altres tests són el test de *stencil* o el *pixel ownership*. Les operacions de fragment inclouen *alpha blending* i les operacions lògiques per a combinar el nou fragment amb el que ja existia a la mateixa posició del *Framebuffer*.

9. El resultat final un cop s'ha enviat i processat la geometria el trobem al *Framebuffer*, que podrem mostrar per pantalla o fer servir com a textura per a un altre pas de pintat.

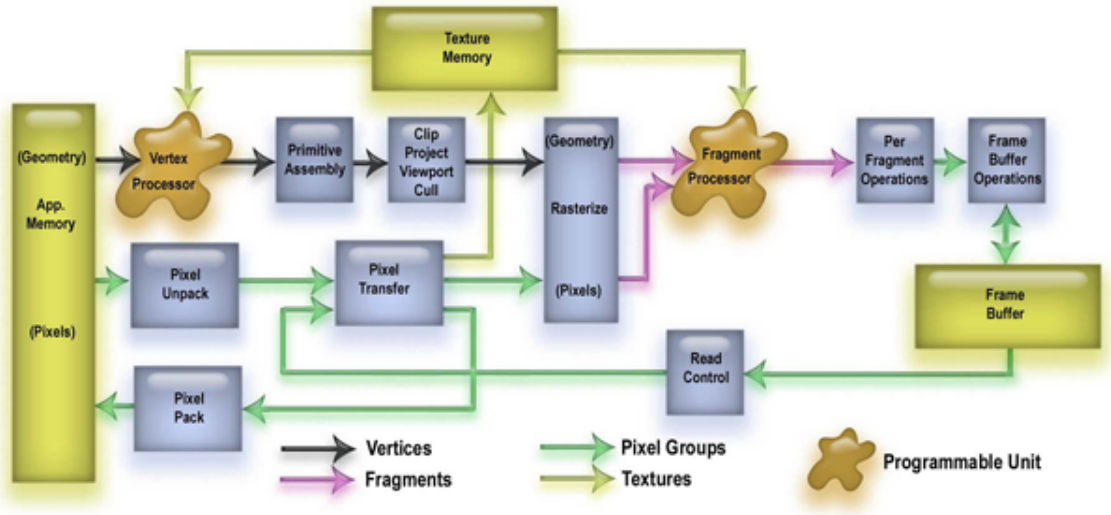

Figura A.1: Pipeline d'OpenGL 2.1. No es mostra el *geometry shader*, que hauria de ser una etapa programable situada entre les fixes de *primitive assembly* i *clipping*. *Imatge extreta de [37].*

Amb el nou hardware gràfic, que inclou un Tessel·lador hardware, la versió 4.1 d'OpenGL afegeix dos nous shaders, el *tessellation control shader*, que calcularà els paràmetres d'entrada al Tessel·lador, i el *tessellation evaluation shader*, que acabarà de definir les posicions i altres atributs dels vèrtexs generats pel Tessel·lador. El Tesel·lador, que ens permetrà subdividir la geometria d'entrada, s'executarà si i només si algun dels dos tipus de *shader* esmentats està actiu. Aquestes noves etapes se situen entre la de *primitive assembly* i el *geometry shader*.

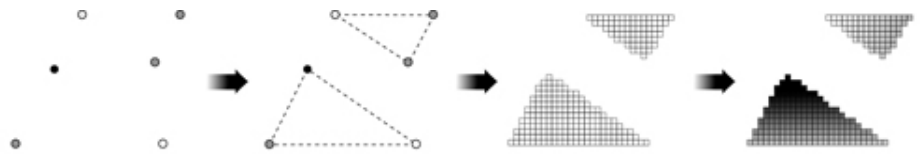

Figura A.2: Exemple de processat al *pipeline*: vèrtexs d'entrada, primitives, fragments rasteritzats i fragments amb el color final calculat. *Imatge extreta de [37].*

Apèndix B

CUDA

CUDA és l'acrònim de *Compute Unified Device Architecture* [27]. Es tracta d'una arquitectura hardware i software de NVIDIA que permet a les GPU executar programes escrits en C, C++, Fortran, OpenCL, DirectCompute i altres llenguatges de programació. L'objectiu és poder realitzar programació de caràcter general aprofitant el paral·lelisme i la potència de càlcul dels dispositius gràfics actuals.

Un programa escrit en CUDA crida a *kernels*, funcions que voldrem que s'executin en paral·lel. Els *kernels* s'invoquen des del *host* (per exemple, la CPU) i s'envien a executar als *devices* (per exemple, la targeta gràfica).

Cada *thread* correspondrà a una instància d'un d'aquests *kernels*. Un *thread* està format per un identificador, el seu comptador de programa, memòria local, registres, dades d'entrada i resultats de sortida. Els registres són el tipus de memòria més ràpida que podem tenir. A més, un *thread* també pot accedir a memòria local privada. No obstant, com l'espai per a la memòria local forma part de la memòria global, els accessos a aquesta seran dels més lents.

Els *threads* s'agrupen el *blocks*. Els *threads* d'un mateix bloc poden cooperar entre ells mitjançant sincronisme i memòria compartida pel bloc. La memòria compartida és molt més eficient que la memòria global si tots els *threads* d'un bloc hi accedeixen alhora i a posicions properes, ja que s'hi portaran totes les dades necessàries de cop des de la memòria global. Per això, és útil poder sincronitzar entre els diferents *threads* del bloc quan fan els accessos, i és bo intentar que aquests estiguin ben definits per a tenir coherència espacial a les dades.

Un bloc pot estar dividit en més d'un *warp*. Cada *warp* és un grup de *threads* executats en mode SIMD, és a dir, que tots ells comparteixen la mateixa instrucció i cadascun l'aplicarà a les seves dades. Com a mínim, un *warp* ha de tenir 32 *threads*. Degut a que l'execució dels diferents *warps* d'un bloc serà serialitzada per a poder aplicar el model SIMD, els codis amb molts condicionals i amb molta divergència de camins d'execució no aconseguiran un rendiment tant bo. Un clar exemple de codi molt divergent és la cerca d'interseccions d'un raig en estructures de particionat de l'espai.

Finalment, els *blocks* s'agrupen en arrays que s'anomenen *grids*. Entre els diferents blocs d'una mateixa *grid* s'han de comunicar a través de la memòria global, que és el tipus més lent de memòria.

Hi ha dos tipus més de memòria per a accessos de només textura des del *device*: la memòria constant i la memòria de textura. A diferència de les altres memòries, les dades

d'aquestes poden portar-se a la caché per accelerar els accessos. La memòria constant l'escriu el *host* abans de cridar al *kernel*. Si tots els *threads* accedeixen a una mateixa posició de memòria i les dades son a la caché, en un cicle es rebrà la petició. Altrament, se serialitzaran els accessos. La memòria de textura és especialment útil, a part de per tenir textures, per a accessos pels quals sabem que hi haurà molta localitat espacial.

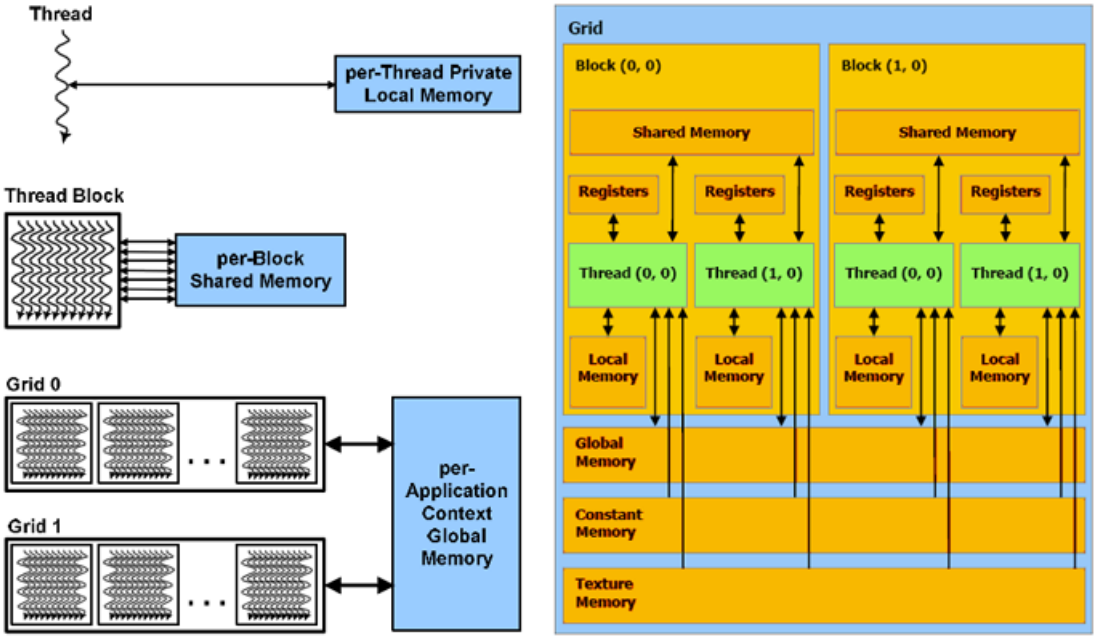

Figura B.1: Diagrama de l'arquitectura de *threads*, *blocks* i *grids* de CUDA. La part dreta mostra els diferents tipus de memòria i com són accedits pels diferents nivells de la jerarquia de *threads*. *Imatges extretes de [13].*

El codi de la figura B.2 mostra un *kernel* molt senzill de CUDA que rep com a entrades dos vectors, `A` i `B` i calcula el vector suma resultant a `C`. Observem que per a calcular l'índex `i` al qual haurà d'accedir cada *thread* multipliquem l'identificador de bloc pel nombre de *threads* que té cada bloc i li sumem l'identificador de *thread* dins del bloc. El nombre de blocs i de *threads* per bloc és una configuració que passem des del *host* quan es fa la invocació del *kernel*. Aquest valor no fa falta que sigui conegut en temps de compilació del programa principal.

Una altra observació sobre el *kernel* és el fet d'accedir al camp `x` dels identificadors, degut a que podem estructurar els *threads* en més d'una dimensió. També és important comprovar que no ens sortim de l'espai reservat al vector, ja que això pot passar si la mida `N` no és múltiple del nombre de *threads* per bloc.

```
// Variables dels arrays: h_ per a host i d_ per a device
float* h_A, h_B, h_C;
float* d_A, d_B, d_C;

// kernel de CUDA
__global__ void SumaVectors(const float* A, const float* B, float* C, int N)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}

// programa principal que crida al kernel
int main() {

    /* ... */

    // Reservem espai pels vectors a memoria del device
    cudaMalloc((void**)&d_A, size);
    cudaMalloc((void**)&d_B, size);
    cudaMalloc((void**)&d_C, size);

    // Copiem els vectors d'entrada de memoria de host al device
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Invoquem el kernel
    int threadsPerBlock = 256;
    int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
    SumaVectors<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);

    // Copiem el vector resultant del device al host
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    /* ... */
}
```

Figura B.2: Exemple de programa en CUDA que suma dos vectors a la GPU.

Apèndix C

Manual d'usuari

Per a iniciar el programa només cal obrir l'executable. Automàticament, es carregarà el model petit del centre de Barcelona i les seves textures assignades, a la vegada que s'iniciarà OpenGL i OptiX. Aquest procés triga uns 15 segons. Un cop s'hagi iniciat del tot, podrem començar a navegar pel model, canviar opcions o carregar un nou model. La figura C.1 mostra l'aspecte de la interfície en aquest moment.

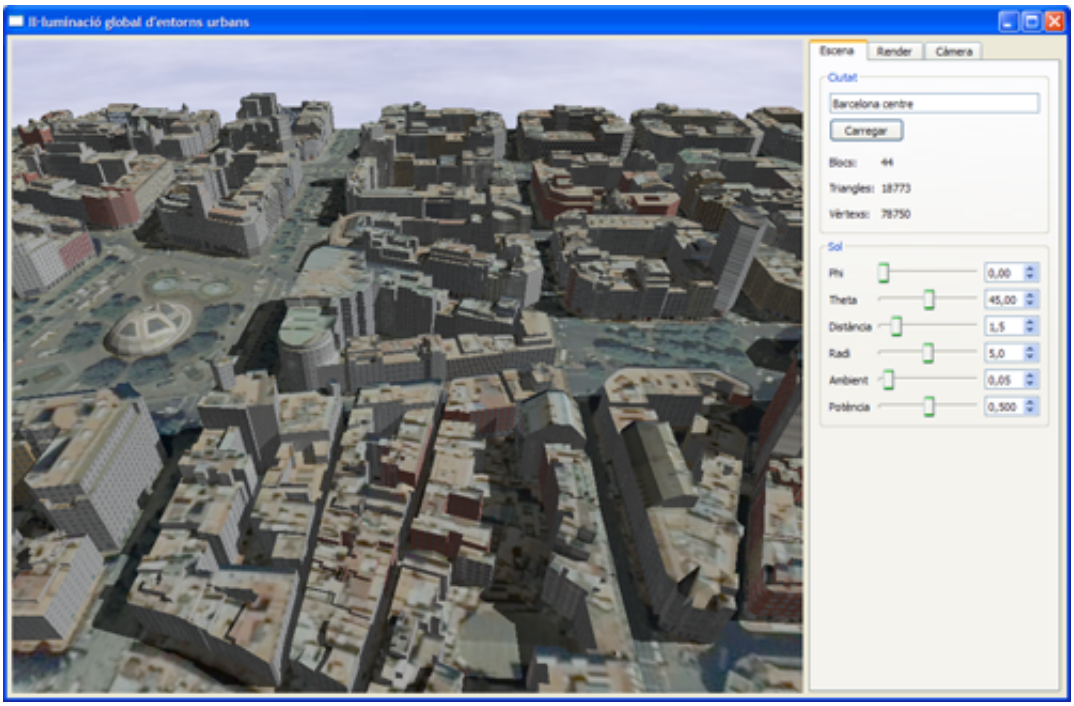

Figura C.1: Interfície del visualitzador.

Si tenim el focus al *widget* d'OpenGL, podrem realitzar diverses **accions amb el teclat i ratolí** per a navegar més còmodament. Concretament:

- *W, A, S, D*: moviment de la càmera endavant (W), enrere (S), cap a l'esquerra (A) i cap a la dreta (D).
- *Botó esquerre del ratolí*: deixant-lo premut i movent amunt i avall o esquerra i dreta el ratolí canviarem els angles amb l'eix X o Y de la càmera, respectivament.
- *Control + Botó esquerre del ratolí*: amb els dos premuts alhora i movent el ratolí fent un cercle al voltant del centre de la imatge, podem ajustar l'angle Z de la càmera, és a dir, la rotació del vector vertical.

- *Shift + Botó esquerre del ratolí*: amb els dos premuts alhora i movent el ratolí amunt i avall podem modificar l'angle de la càmera per a fer *zoom*.
- *Botó dret del ratolí*: deixant-lo premut movent el ratolí podem moure el VRP per a fer *pan*.

A part, la interfície del programa consta de tres pestanyes per a modificar les diferents opcions. La figura C.2 mostra l'aspecte de les tres pestanyes.

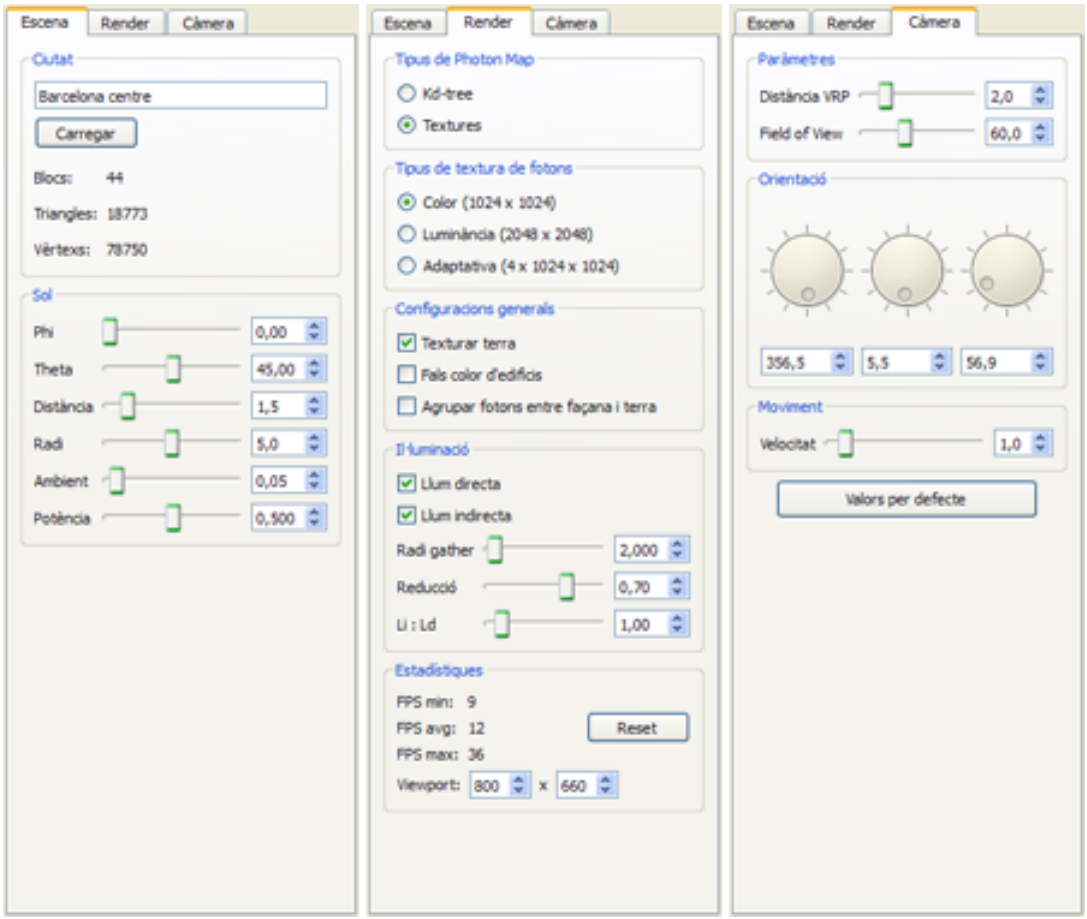

Figura C.2: Les diferents pestanyes d'opcions, d'esquerra a dreta: escena, render i càmera.

- **Pestanya Escena**: trobarem les opcions relacionades amb el model i amb la posició de la llum, és a dir, del Sol.
  - *Botó Carregar*: ens obrirà un diàleg per a seleccionar el model que volem carregar.
  - *Informació sobre la ciutat*: es mostra el nom del model carregat i el nombre d'edificis, triangles i vèrtexs.
  - *Phi, Theta*: angles $\varphi \in [0, 360]$ i $\theta \in [0, 90]$ que defineixen la posició del Sol en coordenades polars.
  - *Distància, Radi*: aquests dos paràmetres ens poden ajudar a focalitzar els fotons sobre una part en concret de l'escena.
  - *Ambient*: permet canviar el coeficient ambient, entre 0 i 1. Es recomana com més baix millor.

  - *Potència*: modifica la potència de la llum, afectant tant a la llum directa com al flux dels fotons.

- **Pestanya Render**: des d'aquí podrem ajustar els diferents paràmetres de la visualització i seleccionar el tipus d'algorisme a fer servir.

  - *Tipus de Photon Map*: podrem alternar entre l'algorisme amb el *kd-tree* i el nostre amb les textures.
  - *Tipus de textura de fotons*: si tenim com a tipus de Photon Map seleccionat el de les textures, podrem canviar entre la textura en color, la de luminància i les adaptatives.
  - *Configuracions generals*: permet activar o desactivar configuracions del render de l'escena, com ara fer servir o no la textura del terra, assignar un color a cada edifici o agrupar fotons entre façanes i terres.
  - *Il·luminació*: tenim opcions que ens permeten desactivar la visualització de la llum directa o indirecta, modificar el radi r^2 per a cercar fotons al pas d'il·luminació, modificar el factor α de reducció del radi a cada iteració i assignar la relació entre llum directa i indirecta.
  - *Comptadors de FPS*: es mostren dades relatives als *frames per segon* mínims, màxims i la mitjana dels últims segons.
  - *Botó Reset*: posa a 0 els tres comptadors de FPS.
  - *Resolució*: permet ajustar de manera precisa l'ample i alt de la imatge que visualitzem.

- **Pestanya Càmera**: ens permetrà modificar la càmera, a vegades de manera més còmoda o precisa que amb teclat i ratolí.

  - *Distància al VRP*: modifica la distància entre VRP i observador, expressada com a multiplicador del radi de l'escena.
  - *Field of View*: modifica l'angle d'obertura vertical de la càmera, entre 5 i 150 graus.
  - *Orientació*: permet definir els tres angles de la càmera per a fixar una orientació.
  - *Velocitat*: defineix el multiplicador de la velocitat de moviment de la càmera.

Apèndix D

Glossari

Atles de textura: textura 2D que conté un munt de textures menors.

BRDF: acrònim de *Bidirectional Reflectance Distribution Function*, funció que simplifica la BSSRDF assumint que el punt d'incidència i reflexió són el mateix. Es defineix com la relació entre la radiància reflectida i la irradiància. És la funció que es fa servir habitualment per a modelar materials.

BSDF: acrònim de *Bidirectional Scattering Distribution Function*, funció BRDF definida sobre tota l'esfera de direccions al voltant d'un punt. Es pot considerar com la combinació d'una BRDF i una BTDF.

BSSRDF: acrònim de *Bidirectional Scattering Surface Reflectance Distribution Function*, funció que relaciona la radiància reflectida en un punt i en una direcció determinada amb el diferencial de flux incident a un altre punt i direcció.

BTDF: acrònim de *Bidirectional Transmittance Distribution Function*, funció similar a la BRDF usada per definir les transmissions a través del material en comptes de les reflexions.

Càustica: fenomen produït per la concentració dels rajos de llum sobre una zona concreta d'una superfície a causa de reflexions i refraccions.

Color bleeding: propagació difosa del color entre superfícies properes.

CUDA: acrònim de *Compute Unified Device Architecture*, arquitectura hardware i software de NVIDIA que permet a les GPU executar programes escrits en C, C++, Fortran, OpenCL, DirectCompute i altres llenguatges de programació.

Emitància radiant: flux emès per unitat de superfície.

Energia radiant: energia total d'un conjunt de fotons de qualsevol longitud d'ona de l'espectre.

Equació de render: equació que ens descriu la distribució de la radiància a una escena, donats un punt i una direcció.

Equacions de Fresnel: equacions que ens permeten determinar la reflectància i transmitància de la llum quan arriba a una superfície, en funció de l'angle d'incidència.

Flux o potència radiant: quantitat total d'energia que travessa una superfície per unitat de temps.

Frame: cadascuna de les imatges o fotogrames amb els quals està composta una animació o visualització interactiva.

Frames per segon (fps): freqüència, nombre de fotogrames que es mostren cada segon.

GPU: acrònim de *Graphics Processing Unit*, processador dedicat a operacions gràfiques i en coma flotant.

Il·luminacio global: tipus de model d'il·luminació que té en compte la resta d'objectes de l'escena i

Il·luminacio local: tipus de model d'il·luminació que no té en compte la resta d'objectes de l'escena, de manera que no pot simular ombres o reflexions, entre d'altres.

Intensitat radiant: flux per unitat d'angle sòlid.

Irradiància: flux incident per unitat de superfície.

Kd-tree: estructura de dades útil per a particionar un espai k-dimensional fent servir plans paral·lels als eixos de coordenades. És molt eficient per a cerques de proximitat i de rangs.

Medi no participatiu: medi buit que no afecta a la propagació de la llum, de manera que es propaga en línia recta i sense alterar el flux.

Medi participatiu: tipus de medi no buit que afecta a la propagació de la llum, produint fenòmens d'absorció, emissió o dispersió que alteren el flux transportat o la direcció.

OpenGL: especificació estàndard que defineix una API multillenguatge i multiplataforma per a desenvolupar aplicacions gràfiques.

OptiX: motor de Ray Tracing programable i de propòsit general desenvolupat per NVIDIA.

Ortofotografia: fotografia d'un terreny en la que s'ha aconseguit corregir la perspectiva i mostrar els elements en projecció ortogonal.

Photon Map: literalment, mapa de fotons. Estructura que fa servir l'algorisme de Photon Mapping per a emmagatzemar informació sobre les interaccions dels fotons amb l'escena per a poder calcular posteriorment la il·luminació.

Photon Mapping: algorisme d'il·luminació global que realitza un primer pas de traçat i propagació de fotons, els guarda a una estructura de dades i els fa servir posteriorment al segon pas per a il·luminar les superfícies visibles que obté amb un traçat de rajos.

Radiància: flux per unitat d'angle sòlid i de superfície projectada.

Radiositat: sinònim d'emitància radiant.

Radiosity: família d'algorismes d'il·luminació global que es basen en el mètode dels elements finits per a solucionar l'equació de render i generar una imatge realista.

Raig primari: raig que s'emet directament des de la càmera.

Raig secundari: raig que s'emet des d'una superfície després que un raig primari o un altre secundari hi arribi. Si es fa servir per a determinar visibilitat de les fonts de llum s'anomena *Shadow ray*.

Ray Tracing: família d'algorismes d'il·luminació global que es basen en el traçat de rajos i el mostreig aleatori per a solucionar l'equació de render.

Raytracer: literalment, traçador de rajos. Programa que implementa algun algorisme de Ray Tracing per a generar una imatge.

Render: terme anglès que es fa servir per a referir-se al procés de generació d'una imatge. També es fa servir per a anomenar la imatge resultant d'aquest procés.

RGB, RGBA: espai de color que fa servir tres components: vermella (R), verda (G) i blava (B) per a representar els diferents colors. Opcionalment, es pot fer servir una component de transparència, anomenada *alpha* (A).

Subsurface scattering: mecanisme de transport de la llum en el qual penetra dins d'una superfície, es dispersa a causa de les interaccions amb el material i surt de la superfície per diferents punts. És habitual en materials com la cera, el marbre o la pell.

Superfície difosa: superfície que reflecteix la llum incident en totes les direccions per igual.

Superfície especular: superfície que reflecteix la llum incident en una direcció específica.

Texture array: vector lineal de textures. Es pot veure com una textura 3D on la 3a coordenada es fa servir com a índex del vector i no es realitza interpolació entre textures consecutives.

YUV: espai de color que fa servir tres components: luminància (Y) i dues de crominància (U, V) per a representar els diferents colors.

Bibliografia

[1] Aila, T. i S. Laine: *Understanding the efficiency of ray traversal on GPUs.* Dins *Proceedings of the Conference on High Performance Graphics 2009*, pàgines 145–149. ACM, 2009.

[2] Akenine-Möller, T., E. Haines i N. Hoffman: *Real-Time Rendering.* A K Peters, 3a edició, 2005.

[3] Arvo, J.: *Backward ray tracing.* Dins *In ACM SIGGRAPH'86 Course Notes-Developments in Ray Tracing*, 1986.

[4] Benthin, C., I. Wald, M. Scherbaum i H. Friedrich: *Ray tracing on the cell processor.* Dins *Interactive Ray Tracing 2006, IEEE Symposium on*, pàgines 15–23. IEEE, 2006.

[5] Bigler, J., A. Stephens i S.G. Parker: *Design for parallel interactive ray tracing systems.* Dins *Proceedings of the IEEE Symposium on Interactive Ray Tracing*, pàgines 187–195, 2006.

[6] Bikker, J.: *Real-time ray tracing through the eyes of a game developer.* Dins *Interactive Ray Tracing, 2007. RT'07. IEEE Symposium on*, pàgines 1–10. IEEE, 2007.

[7] Carr, N.A., J.D. Hall i J.C. Hart: *The ray engine.* Dins *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pàgines 37–46. Eurographics Association, 2002.

[8] Caustic Graphics: *OpenRL SDK Documentation.*

[9] Cook, R.L., T. Porter i L. Carpenter: *Distributed ray tracing.* Dins *ACM SIGGRAPH Computer Graphics*, volum 18, pàgines 137–145. ACM, 1984.

[10] Dietrich, A., I. Wald, C. Benthin i P. Slusallek: *The OpenRT Application Programming Interface Towards A Common API for Interactive Ray Tracing.* 2003.

[11] Djeu, P., W. Hunt, R. Wang, I. Elhassan, G. Stoll i W.R. Mark: *Razor: An Architecture for Dynamic Multiresolution Ray Tracing.* Report tècnic, The University of Texas at Austin, Department of Computer Sciences, 2007.

[12] Dutré, P., P. Bekaert i K. Bala: *Advanced Global Illumination.* A K Peters, 2a edició, agost 2006.

[13] Farber, R.: *CUDA: Supercomputing for the Masses.* http://drdobbs.com/high-performance-computing/207200659, 2008-2010.

[14] Foley, T. i J. Sugerman: *KD-tree acceleration structures for a GPU raytracer.* Dins *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pàgines 15–22. ACM, 2005.

[15] Forés, A., S.N. Pattanaik, C. Bosch i X. Pueyo: *BRDFLab: A general system for designing BRDFs.*

[16] Goral, C.M., K.E. Torrance, D.P. Greenberg i B. Battaile: *Modeling the interaction of light between diffuse surfaces.* Dins *ACM SIGGRAPH Computer Graphics*, volum 18, pàgines 213–222. ACM, 1984.

[17] Gunther, J., S. Popov, H.P. Seidel i P. Slusallek: *Realtime ray tracing on GPU with BVH-based packet traversal.* 2007.

[18] Hayward, K.: *Instant Radiosity using OptiX and Deferred Rendering.* http://graphicsrunner.blogspot.com/2011/03/instant-radiosity-using-optix-and.html, 2011.

[19] Jensen, H.W.: *Global illumination using photon maps.* Rendering Techniques, 96:21–30, 1996.

[20] Jensen, H.W.: *Realistic Image Synthesis Using Photon Mapping.* A K Peters, 2a edició, juliol 2001.

[21] Kajiya, J.T.: *The rendering equation.* ACM SIGGRAPH Computer Graphics, 20(4):143–150, 1986.

[22] Keller, A.: *Instant radiosity.* Dins *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pàgines 49–56. ACM, 1997.

[23] Kriványek, J., P. Gautron, G. Ward, H.W. Jensen, E. Tabellion i P.H. Christensen: *Practical Global Illumination with Irradiance Caching.*

[24] Lafortune, E.P. i Y. Willems: *Bi-directional path tracing.* Dins *Compugraphics' 93*, pàgines 145–153, 1993.

[25] Lauterbach, C., M. Garland, S. Sengupta, D. Luebke i D. Manocha: *Fast BVH construction on GPUs.* Dins *Computer Graphics Forum*, volum 28, pàgines 375–384. Wiley Online Library, 2009.

[26] Ludvigsen, H. i A.C. Elster: *Real-Time Ray Tracing Using Nvidia OptiX.* Eurographics, 2010.

[27] NVIDIA: *NVIDIA CUDA Compute Unified Device Architecture: Programming Guide.*

[28] NVIDIA: *NVIDIA OptiX Ray Tracing Engine: Programming Guide.*

[29] Parker, S., W. Martin, P.J. Sloan, P. Shirley, B. Smits i C. Hansen: *Interactive ray tracing.* Dins *Proceedings of the 1999 symposium on Interactive 3D graphics*, I3D '99, pàgines 119–126. ACM, 1999.

[30] Parker, S., M. Parker, Y. Livnat, P.P. Sloan, C. Hansen i P. Shirley: *Interactive ray tracing for volume visualization.* Visualization and Computer Graphics, IEEE Transactions on, 5(3):238–250, 1999.

[31] Parker, S.G., J. Bigler, A. Dietrich, H. Friedrich, J. Hoberok, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison i M. Stich: *OptiX: a general purpose ray tracing engine.* ACM Transactions on Graphics, juliol 2010.

[32] Pharr, M. i G. Humphreys: *Physically Based Rendering.* Morgan Kaufmann, 2a edició, juliol 2010.

[33] Purcell, T.J., I. Buck, W.R. Mark i P. Hanrahan: *Ray tracing on programmable graphics hardware.* Dins *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '02, pàgines 703–712. ACM, 2002.

[34] Purcell, T.J., C. Donner, M. Cammarano, H.W. Jensen i P. Hanrahan: *Photon mapping on programmable graphics hardware.* Dins *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pàgines 41–50. Eurographics Association, 2003.

[35] Rost, R.J. i B. Licea-Kane: *OpenGL Shading Language.* Addison-Wesley Professional, 3a edició, juliol 2009.

[36] Shih, M., Y.F. Chiu, Y.C. Chen i C.F. Chang: *Real-Time Ray Tracing with CUDA.* Algorithms and Architectures for Parallel Processing, pàgines 327–337, 2009.

[37] Shreiner, D.: *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1.* Addison-Wesley Professional, 7a edició, juliol 2009.

[38] Stich, M., H. Friedrich i A. Dietrich: *Spatial splits in bounding volume hierarchies.* Dins *Proceedings of the Conference on High Performance Graphics 2009*, pàgines 7–13. ACM, 2009.

[39] Suykens - De Laet, F.: *On Robust Monte Carlo Algorithms for Multi-pass global Illumination.* Tesi de Doctorat, Katholieke Universiteit Leuven, setembre 2002.

[40] Szirmay-Kalos, L., L. Szécsi i M. Sbert: *GPU-Based Techniques for Global Illumination Effects.* Synthesis Lectures on Computer Graphics and Animation, 2(1):1–275, 2008.

[41] Veach, E. i L.J. Guibas: *Metropolis light transport.* Dins *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pàgines 65–76. ACM, 1997.

[42] Wald, I., P. Slusallek, C. Benthin i M. Wagner: *Interactive rendering with coherent ray tracing.* Dins *Computer Graphics Forum*, volum 20, pàgines 153–165, 2001.

[43] Ward, G.J., F.M. Rubinstein i R.D. Clear: *A ray tracing solution for diffuse interreflection.* Dins *Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '88, pàgines 85–92. ACM, 1988.

[44] Whitted, T.: *An improved illumination model for shaded display.* Communications of the ACM, 23(6):343–349, 1980.

[45] Woop, S., J. Schmittler i P. Slusallek: *RPU: a programmable ray processing unit for realtime ray tracing.* Dins *ACM SIGGRAPH 2005 Papers*, pàgines 434–444. ACM, 2005.